Towards a Formal Semantics for System Calls in terms of Information Flow

Laurent Georget Guillaume Piolle
Frédéric Tronel Valérie Viêt Triem Tong
EPC CIDRE SUPELEC/INRIA/CNRS/University of Rennes 1
Rennes, France
Email: *first_name.last_name*@supelec.fr

Mathieu Jaume
Team MoVe, UPMC LIP6
Paris, France
Email: mathieu.jaume@lip6.fr

***Abstract*—We propose a new semantics for system calls, which focuses on the information flows they generate in a UNIX OS. We built a prototypal model of an OS and system calls using the concurrent transaction logic along with its interpreter. We have yet a few results and applications that show the usefulness of our semantics to model an OS from a kernel point of view. Once completed, we expect our semantics to enable us to extensively test security software implemented inside the kernel, among other use cases.**

***Keywords*—Operating Systems; Security; System Calls; Information Flow.**

I. Introduction

In an operating system (OS), system calls define a clear boundary between the kernel-land, where lives the core part of the OS and the userland, which contains all the end-user applications. While the kernel runs with all privileges on the hardware, the user applications are granted only few rights to prevent them from interfering with each other. When such applications want to perform tasks requiring an access to the hardware or to communicate with each other, they need to ask the kernel for this service by the mean of *system calls*. Those are the only code interfaces between kernel-land and userland.

System calls are necessary because they enable the userland processes to achieve complex tasks. By allowing processes to communicate with each other and storing data, they let information flow in the system, which is necessary for all kinds of tasks. However, this can also be problematic because OS security mechanisms mostly rely on access control which can only prevent access to containers of information, and not to the information itself. Therefore, once a piece of information has left its original container, it is difficult to control the way it is accessed. One way to keep being able to know where each piece of information is in the system is to monitor the information flows using meta-information attached to each container of information called *taints*. Each time information flow from one container to another, the receiver gets *tainted* with the meta-information from the source. This way, each container is tainted accordingly to the origin of the data it contains. This monitoring can be performed at several levels of granularity depending on what is considered an elementary container of information. In our case, we are interested in OS-level containers, such as processes, files, sockets, etc. As the entire OS security relies on the correct interpretation of the information flows, information flows monitors are critical, and it is important to know how far they can be trusted.

Our main problem is that it is hard to know if information flows monitors actually interpret the flows correctly and if their view of the system is consistent with the actual state of the system. A formal proof through static analysis for example is infeasible if they are implemented on top of a preexisting OS which was not designed for proof. To tackle this issue, we need to know what are the information flows caused in an OS. Our work is based on the two following hypothesis. First of all, only processes cause information flows in a system because they are the active entities that execute codes whereas the other components are passive. Second, system calls are necessary for all information flows deliberately caused by processes. This leads us to define more precisely what the information flows generated by each system call are. Our goal is a formalized semantics of system calls in terms of information flows for a preexisting UNIX-like OS. This semantics will define clearly and unambiguously what information flows can be caused in the system and by which means. This would give us a reference to state on the correctness of a given information flow monitor. The rest of this paper is organized as follows. In Section I, we present the state of the art. In Section II, we discuss our working hypothesis and we present the main elements of our modelization. In Section III, a commented example of a system call gives the intuition of our syntax. Then, in Section IV, we present our work on a use case of our semantics: testing an existing information flow monitor. We conclude and give perspectives in Section VI.

II. Related Work

Information flow control systems is a long-studied topic in security. Some are implemented on top of preexisting OSes, which makes them more likely to be used in practical applications than *ad hoc* solutions. Blare [1] is built for the mainstream Linux kernel. Contrary to other tools such as Flume [2], it monitors automatically all the processes and does not rely on a user-land daemon to intercept system calls. Blare is based on the Linux Security Modules (LSM) framework [3]. LSM adds security fields in existing data structures inside the kernel and provides security developers with hooks. Those

 ISBN: 978-1-61208-399-5

are placed before any access to a kernel object, which can represent a file for example. Functions can be set up on any hook so that when it is triggered the function is executed to mediate the access to the object. Blare plugs in functions in charge of calculating the propagation of taints on the kernel objects. The correctness of security softwares based on LSM relies on the correct placement of the hooks, otherwise, the access control is flawed. Several approaches using static analysis have been proposed so far to verify respectively that (1) all security-sensitive operations are mediated by a hook [4], and (2) to check that the necessary set of hooks is called before each security-sensitive operation on a kernel object [5]. Although those works are necessary to ensure that security mechanisms based on LSM are correct, they are not sufficient because they cannot be used to verify those mechanisms themselves. Our work aims at ensuring that information flows are correctly interpreted by security mechanisms, in order to give stronger guarantees on the validity of their model and the correctness of their implementation.

III. Modeling information flows in an OS

A. Discussion on our Working Hypothesis

Our working hypothesis is that no information flow can occur in an OS without at least the execution of one system call. Therefore, the kernel that receives the system calls and acts upon them is aware of all the information flows in the system. This hypothesis arises from the fact that system calls are the only way for a process to ask the kernel to run code on its behalf, and only code run by the kernel, that has high privileges, can make information flow between processes or from processes to other containers like files or network sockets. Of course, processes are free to manipulate the information they have inside their own memory space and to perform arbitrary manipulation on it. Hence, we have to consider containers of information at a coarser grain than individual memory locations. The file descriptor abstraction used under UNIX is useful here. On a first approximation, for our prototype, we consider as a container a process or anything that can be handled by a file descriptor in a UNIX abstraction. This covers files, network sockets, message queues, etc.

Our hypothesis has some drawbacks, too. For example, when two processes want to share a memory area, they need only a few system calls to set up and map the memory area and then, they can communicate and exchange information freely without any additional system call. So, to fully capture the information flows in this case, we have to overapproximate them and consider that the processes have exchanged *all* the information they had while the memory remained shared. A few malicious processes could then abuse this overapproximation by setting up shared memory areas and opening a lot of files to mask their real objective. Typically, a virus that would infect a lot of core processes could lower the accuracy of the information flow monitor so much that it would become practically ineffective. However, this would only result in valid flows being denied, not illegal flows being granted.

B. Scope and Object of our Modelization

To describe the information flows caused by each individual system call, we need to consider their environment. Information flows are caused by processes and occur between containers. The internal state of the OS also matters because the results of some system calls depend on specific conditions which are out of control from the process. For example, there is a limit on the total number of files that can be open simultaneously in an OS. Therefore, the same system call with the same parameters issued by a process asking for a given file to be opened can succeed or fail in two contexts indistinguishable from the process's point of view. This fact leads us to model the entire OS, with its internal state along with the system calls. Unlike many models focusing on processes like the process algebras which describe communication between them, we consider the kernel's point of view.

For our work, we first focused on the MINIX OS [6]. This OS is a fully-featured UNIX environment with a very clear and simple design. The methodology used to build our prototype can be extended to others kernels with a larger codebase such as Linux. In our model, the kernel is a database of containers of information. These containers may be of various types: files, network sockets, processes, memory segments, etc. For each of these containers, we maintain various pieces of information such as the size in case of a file, the program being executed in case of a process, etc. Separately, we have a list of observed information flows. We model system calls as transactions impacting the database asynchronously. Each time a system call is triggered, it is executed, objects in the database are created, altered or even deleted, and new information flows may be added to the list. This is where our semantics of system calls is necessary; without it, our model could not be proved accurate and we may miss important side effects in the internal state of the kernel (represented in our database), or even information flows. We also made our semantics and model executable. This is important because it allows us to build runnable test cases.

C. Model of the System Calls and the Database

As we previously said, system calls are modeled as transactions asynchronously reaching the database. Those transactions may alter the database in any possible way but they are executed as a whole, i.e., either the transaction is entirely executed or it is not at all. Furthermore, two system calls transactions cannot interfere with each other: their execution may be interleaved but not simultaneous and they cannot access the same objects if they are interleaved. These precautions are necessary to keep the database in a valid state, consistent with a real kernel. Of course, in a real multiprocesses system, two system calls may be executed at the same time but synchronization mechanisms, such as locks, prevent them from accessing the same objects and data structures in the kernel at the same time so our model remains valid if we make the assumption that synchronization is correctly enforced in the real system. If this were not the case, the real system would

 ISBN: 978-1-61208-399-5

TABLE I. FILES' META-ATTRIBUTES.

Name	Meaning
path	File system path
rd_locked	Read lock
wr_locked	Write lock
uid	Owner's id
gid	Owning group's id
mode	Access rights and flags `setuid`, `setgid`, `sticky bit`

TABLE II. FILES DESCRIPTORS' META-ATTRIBUTES.

Name	Meaning
file	File corresponding to the file descriptor
procs	Processes owning this file descriptor
opts	Read-only, Write-only, Read-Write, Append, etc.
pos	Current position in the file

be buggy and its internal state inconsistent, so no model would be accurate.

To model the system calls, we used the Concurrent TRansaction logic, CTR, by Bonner and Kifer [7]. This logic provides us with both a syntax and a semantics to express changes and consultations in a database as transactions. The database, in the case of the CTR, can have an arbitrary structure, provided that it can be accessed by the means of clearly defined logical primitives. In our case, those primitives are the consultation of the database, the atomic change of an attribute of a single object in the database, the creation of a new object in the database, the deletion of an object, and the registration of a newly detected information flow. As its name suggests, the CTR lets two or more transactions run concurrently, in an interleaved manner, which is desired in our case. Last but not least, the CTR provides us with an executional semantics: an interpreter can be built to simulate the execution of a transaction in a database. The documentation of our interpreter is available online [8].

Our database model contains four tables and a list. The tables are (1) the table of active processes in the system, (2) the table of existing files in the system, (3) the table of open file descriptors, and (4) the table of memory areas allocated to processes. The list contains the information flows the interpreter detects. It is chronologically sorted. This is of course a small model built only for the sake of prototyping because we consider only processes and files as containers of information but it can be extended. Each table contain a set of entries. An entry is indexed by a unique identifier and represents an object in the OS. For example, an entry in the processes table is a living process in the system. Each entry is described through several attributes. Tables I and II describe respectively the fields contained in the entries of the files table and the file descriptors tables. As one can see, we try to mimic as closely as possible the real OS and we replicate the semantics of `open` and `close` because obviously, information flows are not identical for open and closed files. This is why we made the distinction between files and file descriptors, which correspond to files opened by some process.

IV. EXAMPLE OF A SYSTEM CALL: `READ`

System calls are written as a set of logical clauses. Each clause is a transaction which corresponds to the system call. The clauses are mutually exclusive, only one of them will execute. They correspond to the different behaviors of the system call: there is a clause for each error case and each valid case. The way the interpreter chooses the clause to execute is simple: it tries them one after the other, until one can be entirely executed. Each clause begins with the header of `read` with some arguments. Arguments can be preceded by a $+$ sign or a $-$ sign. A $+$ sign means that the argument is an input, a $-$ sign that it is an output. A system call usually has a return value but can also return values through a pointer passed as an argument. Our notation captures this.

$$\texttt{read}(\text{+FileDescriptor, +Buffer, +Size, -Return}) \leftarrow \quad (1)$$
$$\neg(\texttt{GET\_FID\_FOR\_PROC}(caller, FileDescriptor, Fid)) \quad (2)$$
$$\otimes\ Return = \texttt{EBADF}. \quad (3)$$

This first clause describes the execution of `read` with an invalid file descriptor as first argument. The clause's body starts with a query called `GET_FID_FOR_PROC` which returns, for a given process and a given file descriptor, the file referenced by the file descriptor for the process. This is a logical predicate having logical value `true` if the file descriptor is valid, in which case Fid is a reference to the file and `false` if the file descriptor is invalid. In the first clause, the file descriptor is invalid. What comes after is an addition of the CTR to the predicate calculus. The operator $\otimes$ is the *sequential conjunction*. The intuition of this operator is that if both A and B are transactions, then $A \otimes B$ is the transaction whose execution consists of the correct execution of A *followed by* the correct execution of B. If A has not logical value *true*, which means it cannot be executed, then B is not executed (its effects on the database, if any, are ignored) and the whole transaction $A \otimes B$ is *false* (which can be thought of as *aborted*). If A is true, B is executed and if B is false, the transaction is false too. Here, we test the negation of `GET_FID_FOR_PROC` so if the file descriptor is invalid, the previously undetermined value *Return* is now known to be equal to the constant `EBADF`. If the file descriptor is valid, the clause is rejected and another one is tried. When a clause is rejected, all its effects on the database are *rollbacked*.

$$\texttt{read}(\text{+FileDescriptor, +Buffer, +Size, -Return}) \leftarrow \quad (4)$$
$$\texttt{GET\_FID\_FOR\_PROC}(caller, FileDescriptor, Fid) \quad (5)$$
$$\otimes\ Fid.opts \not\ni \texttt{READ} \quad (6)$$
$$\otimes\ Return = \texttt{EIO}. \quad (7)$$

In the second clause, we first check that the file descriptor is valid (5), then if it is, we check if the file was opened in reading mode (6) and if it is not, we conclude with the

returned value of constant `EIO` (7) (Input/Output Error). If it is, the transaction aborts. Here, we see how the transaction representing the system call refers to the content of the database representing the OS's objects. Fid refers to an entry in the file descriptors table, and $Fid.opts$ refers to the value of the attribute $opts$ inside this entry.

$$\texttt{read}(\text{+FileDescriptor, +Buffer, +Size, -Return}) \leftarrow \quad (8)$$

$$\texttt{GET\_FID\_FOR\_PROC}(caller, FileDescriptor, Fid) \quad (9)$$

$$\otimes\, Fid.opts \ni \texttt{READ} \quad (10)$$

$$\otimes\, Size > 0 \quad (11)$$

$$\otimes\, ReqSize\ is\ min(Size, Fid.file.size - Fid.pos) \quad (12)$$

$$\otimes\, Buffer \ll Fid.file \quad (13)$$

$$\otimes\, Fid.pos := Fid.pos + ReqSize \quad (14)$$

$$\otimes\, Return = ReqSize. \quad (15)$$

Finally, the third clause describes the execution of `read` that actually ends up with an information flow. In this clause, after the first check identical to the previous case (9), the file open mode is tested (10). The size to read is checked to be non-null (11). Then, a new name *ReqSize* is created and immediately given the minimum value between the requested size and the number of bytes left in the file (12). The special syntax that follows means that an information flow took place from the file to *Buffer*, which represents the memory zone where the content of the file is read by the process (13). Next, another syntax element is introduced (14). The current position in the file (which is one of the numerous elements of the database) is *updated* and becomes equal to the old position plus the number of bytes read. Finally, the returned value is the number of bytes read (15).

V. Usage of the Semantics

The semantics for system calls is executable. Sleghel et al. built in 2000 an interpreter for the CTR [9]. Of course, their work was not directly usable for us because they used a very generic model of relational database. In our very specific case, we needed a special database model to account for the particular nature of an OS's kernel. Fortunately, the CTR is not bound to any specific database but can be used with any model as long as elementary operations to query and update the database are provided. So, we built our interpreter basing our implementation on Sleghel et al.'s work for the inference engine and implementing our own elementary predicates. The inference engine, which implements the inference rules of CTR, is written in SWI Prolog [10] and the database part in C++. The latter lets us implement all kinds of side effects in the database and instrument the interpreter. Sleghel et al.'s interpreter was built for another Prolog flavor, but we chose to port it to SWI Prolog because of its handy C++ interface.

The motivation behind the construction of our semantics was to test Blare [1], an information flow monitor. While this is still an on-going work we already have encouraging results on our semantics and interpreter. We are able to set the model of the OS, i.e., the database, in a given initial configuration. Then, we can run simple test cases such as two concurrent processes trying to atomically write in the same file at the same time. One of the processes output gets overriden by the other and the result depends on the order of execution of the writings. Our interpreter lets us effectively see that fact and we can see the content of the database in each situation. This is very interesting because, using the same test case once we get a full formal semantics for Linux system calls, if we reproduce the same situation on an OS equipped with Blare, we should see the same result. If Blare tells us that the file contains information from both processes or information from none of the processes, we can tell it is wrong. Of course, this is a simple example and there are many cases much more complex to deal with.

VI. Conclusion and Future Work

We presented a formal semantics for MINIX system calls for and a methodology to build similar ones for any UNIX-like OS. To the best of our knowledge, this is the first time a semantics is proposed for system calls in preexisting OSes. Our work will be useful to get more confidence in the correct functionning of those information flow monitors. The semantics may also have other uses in the future, such as proofs of correctness for the implementation of system calls. Research work on information flow control is far from being over and more work is needed to achieve a control as accurate as possible, and thus, to bring more security to end-users of OSes. We follow on this track to improve existing solutions and make both OSes and security mechanisms more trustworthy.

References

[1] L. George, V. Viêt Triem Tong, and L. Mé, "Blare tools: a policy-based intrusion detection system automatically set by the security policy," Recent Advances in Intrusion Detection. Saint-Malo, France: Springer Berlin Heidelberg, Sep. 2009, vol. 5758, pp. 355–356.

[2] M. Krohn *et al.*, "Information flow control for standard OS abstractions," ACM Symposium on Operating Systems Principles. Stevenson, WA, USA: ACM, 2007, pp. 321–334.

[3] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman, "Linux security modules: general security support for the linux kernel," USENIX Security Symposium. San Francisco, CA, USA: USENIX Association, 2002, pp. 17–31.

[4] X. Zhang, A. Edwards, and T. Jaeger, "Using CQUAL for static analysis of authorization hook placement," USENIX Security Symposium. San Francisco, CA, USA: USENIX Association, Aug. 2002, pp. 33–48.

[5] T. Jaeger, A. Edwards, and X. Zhang, "Consistency analysis of authorization hook placement in the linux security modules framework," ACM Transactions on Information and System Security, vol. 7, no. 2, May 2004, pp. 175–205.

[6] A. Tanenbaum and A. S. Woodhull, Operating Systems: design and implementation, 3rd ed. Upper Saddle River: Prentice Hall, 2009.

[7] A. J. Bonner and M. Kifer, "Concurrency and communication in transaction logic," Joint Intl. Conference and Symposium on Logic Programming. MIT Press, 1996, pp. 142–156.

[8] L. Georget, "ALFRED, an interpreter for the semantics of system calls," 2014, [Retrieved: 2015.03.02]. [Online]. http://www.lgeorget.eu/alfred/

[9] A. F. Sleghel, "An optimizing interpreter for concurrent transaction logic," Ph.D. dissertation, University of Toronto, 2000.

[10] J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager, "SWI-prolog," Theory and Practice of Logic Programming, vol. 12, no. 1-2, Jan. 2012, pp. 67–96.

 ISBN: 978-1-61208-399-5