CMCAP: Ephemeral Sandboxes for Adaptive Access Control

Theogene Hakiza Bucuti
Department of Computer Science and Engineering
University of North Texas

Ram Dantu
Department of Computer Science and Engineering
University of North Texas

Kirill Morozov
Department of Computer Science and Engineering
University of North Texas

ABSTRACT

We present CMCAP (context-mapped capabilities), a decentralized mechanism for specifying and enforcing adaptive access control policies for resource-centric security. Policies in CMCAP express runtime constraints defined as containment domains with context-mapped capabilities, and ephemeral sandboxes for dynamically enforcing desired information flow properties while preserving functional correctness for the sandboxed programs. CMCAP is designed to remediate DAC's weakness and address the inflexibility that makes current MAC frameworks impractical to the common user. We use a Linux-based implementation of CMCAP to demonstrate how a program's dynamic profile is used for access control and intrusion prevention.

CCS CONCEPTS

• **Security and privacy → Access control**; *Information flow control.*

KEYWORDS

adaptive access control, runtime containment, intrusion prevention, ephemeral sandboxes, information flow control

ACM Reference Format:
Theogene Hakiza Bucuti, Ram Dantu, and Kirill Morozov. 2019. CMCAP: Ephemeral Sandboxes for Adaptive Access Control. In *The 24th ACM Symposium on Access Control Models and Technologies (SACMAT '19), June 3–6, 2019, Toronto, ON, Canada.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3322431.3325414

1 INTRODUCTION

Access control-based intrusion prevention is a mechanism that restricts programs' ambient permissions to prevent unexpected effects when processing untrusted input. It reinforces system security with fine-grained policies to remediate the excessive permissiveness in traditional access control where exploitable bugs in running programs can give remote attackers the effective user's full permissions to local resources. Sound implementations of the mechanism have been integrated in contemporary operating systems in the form of mandatory access control (MAC), system call filtering, and sandboxes. However, despite the common availability of current state-of-the-art protection mechanisms in today's operating systems (such as SELinux [9] and AppArmor [2] on Linux), their adoption remains impractical to the common user because of the level of technical expertise required to make them work.

The inflexibility of common MAC implementations is rooted in expecting users to enforce and maintain fine-grained policies against running programs. The expectation has two main flaws: (1) application developers often lack a "security first" mindset (or even a reasonable interest in system security) when developing programs, and (2) crafting security policies as patches to insecure programs results in inconsistencies due to uncoordinated updates and discrepancies between a program's functional requirements, its security logic, and varying system security requirements.

Notable approaches to avoid the dual coding problem in MAC frameworks include the enforcement of the least privilege principle from within the application. Capability systems such as FreeBSD's Capsicum [7, 14] and Linux' Seccomp allow programmers to reduce the attack surface in their applications by invoking security-specific system calls in their applications to restrict permissible capabilities at runtime. Information flow-based mechanisms like FlumeOS [10] also allow converting programs into security-aware components to minimize system exposure to untrusted code. Both approaches are only effective if applications can be rewritten to fit a given security model, and they complicate policy analysis and maintenance for security administrators [1]. Our approach is to focus on what a security policy protects (system objects) while supporting existing programs in their current form.

CMCAP is an application containment framework based on the principle of decentralized specification and adaptive enforcement of resource-centric access control policies. CMCAP allows the user to group system resources into custom containment domains based on security preferences. Each domain specifies operational contexts determining access modes for protected resources, and hierarchical extensions to existing policies do not require elevated privileges when a derived policy is less permissive than its parent policy. CMCAP's deductive system governs domain transitions based on a program's runtime profile, providing a means to adaptively isolate programs or groups of related programs into restricted namespaces where potential damage is contained. Since security policies are formulated in terms of protected objects rather than the agents accessing them, it is easier for the user to express and verify a desired safety property for a resource of interest regardless of which programs will run on the system. This reduces complexity in policy development or verification, and eliminates inconsistencies caused by program-targeted policies when programs and policies evolve independently.

We implemented CMCAP for Linux based on the LSM interface to take advantage of the already existing security hooks in the kernel, and used it for performance evaluation. Ephemeral sandboxes were designed by combining Linux namespaces and a pseudo-filesystem. They are generated when access to a virtualized resource is received, and do not persist past the execution context of the sandboxed program's instance.

This paper's contributions include:

- The specification of resource-centric policies for access control and information flow tracking on filesystem objects.
- The design of ephemeral sandboxes for automatically isolating program instances whose runtime characteristics contain a forbidden information flow.
- The implementation of CMCAP on Linux as a security module in the kernel, and a companion userspace toolsuite for specifying and inspecting CMCAP policies using logical queries.

The rest of the paper is organized as follows: Section 2 presents a conceptual design for resource-centric policies in CMCAP. A summary of access control semantics in CMCAP is presented in Section 3. A system design and performance evaluation of a Linux-based implementation of CMCAP are demonstrated in Section 4 , followed by a discussion of related work in Section 5. Concluding notes for future development precede the references section at the end.

2 RESOURCE-CENTRIC POLICIES FOR INTRUSION PREVENTION

CMCAP policies create containment domains which define information flow restrictions for groups of passive objects such as files, sockets, and memory segments. Running programs are active objects that may transition into different domains based on their runtime characteristics. When considering access requests, a program's history determines if the access would not violate the policy that applies to the requested resource. Therefore since policies are resource-centric, a program's capabilities change dynamically.

2.1 Context-mapped capabilities

DEFINITION 1. *An operational* `context` *is a containment boundary for the information content in system objects. It serves as a documentation of provenance for data in storage, and an estimation of information propagation in running programs.*

Operational contexts can be regarded as users in a DAC model, except that in CMCAP programs transition between contexts as required by the policies that apply to resources they are requesting. The following operational contexts are basic in CMCAP: '`localCTX`' (for flows from and into system-local objects), '`remoteCTX`' (for anything not considered system-local: network, external media,...). Non-basic contexts can be defined as part of a policy if necessary, and their meanings depend on restrictions applied to them by the policy.

DEFINITION 2. *A CMCAP* `capability` *is a (P, C)-tuple where P is a permission and C is an operational context.*

Basic permissions determine the direction of information flow (`READ`, `WRITE`) while contexts describe containment boundaries (system-local, network/external, program-private). The capabilities are "context-mapped" because a program's operational context determines which permissions can be granted to it. In addition to the basic permissions (`READ`, `WRITE`), CMCAP supports a virtual WRITE permission which differs from the normal WRITE in that its resulting information content is not persistent. We will shorten the mentioned permissions as '`r`', '`w`', and '`w/#`', respectively from now on when presenting policy sketches.

DEFINITION 3. *A containment* `domain` *is a collection of resources with similar information flow restrictions.*

For policy visualization purposes, a containment domain is a directory-like structure in CMCAP: it is specified in terms of resources it protects and capabilities that apply to them. A domain's capabilities form an immutable protection state for any resource covered by the domain.

```
{
  homeDMN:{
    objects:["/home"],
    policy:[
      (r,localCTX), (w,localCTX), (r,remoteCTX)
    ]
  }
}
```

Listing 1: Example domain policy

Consider the containment domain defined in Listing 1: it covers all files under the "/home" directory. Programs with a '`localCTX`' operational context may read from or write to files in the domain, while those with a '`remoteCTX`' context may only read. The domain also forbids transitions that would allow a program to overwrite files in the "/home" directory with content from the internet or external media.

2.2 Policy expression and hierarchical composition

CMCAP was designed to scale well for resource groups. Since CMCAP policies apply to containment domains, resources with the same information flow restrictions can be placed/grouped into the same domain with appropriate access control rules to protect them against unwanted access. Expressing containment rules for a resource therefore requires "mounting" the resource in a containment domain that has the appropriate context-mapped capabilities.

DEFINITION 4. *Let $capset_x$ be the set of capabilities allowed in domain* `x`*, and $objset_x$ the set of object handles enumerated in domain* `x`*'s manifest.*

policy-equivalence *: Domains dom1 and dom2 are considered* `policy-equivalent` *if $capset_{dom1} = capset_{dom2}$.*

permissiveness *: Domain dom1 is considered* `more permissive` *than dom2 if $capset_{dom1} \supset capset_{dom2}$.*

policy coverage *: Let dir_x be the set of objects located under object handle* `x` *following a hierarchical object handle resolution (in the same fashion as UNIX path walking). The* `policy coverage` *of domain* `domX`*, denoted as $polc_{domX}$, is the set of objects characterized with $polc_{domX} = \{\{x\} \cup dir_x \mid x \in objset_{domX}\}$.*

policy composition *: Given two domains* `A` *and* `B` *such that $polc_A \supset polc_B$, domain A is said to be B's* `parent domain`*. In that case domain B is considered an* `extension` *of domain A if B is less permissive than A, otherwise B is invalid.*

policy conflict : *There is a* `policy conflict` *between domains* A *and* B *if* $(polc_A \cap polc_B \neq \emptyset) \wedge (capset_A \triangle capset_B \neq \emptyset)$, *with* $\triangle$ *denoting the set difference.*

Definition 5. *A system's* `effective policy` *is the interaction of all the activated domain policies, taking into account composition and possible conflicts.*

Policy resolution for a resource follows conventional path resolution on UNIX, allowing hierarchical policy composition when restricting subtrees of a given directory. In case of conflict between two domains, the least permissive domain takes precedence in path resolution for common objects. Illustrations for policy composition and conflict are given in listing 2.

```
{
  domA:{
     objects:["/a"],
     policy:[(r, localCTX), (w, localCTX)]
  },
  domB:{
     objects:["/a/b"],
     policy:[(r, localCTX), (w, localCTX)]
  },
  domC:{
     objects:["/a/b/c"],
     policy:[(r, localCTX)]
  },
  domD:{
     objects:["/a/b", "/b", "/c"],
     policy:[(r, localCTX), (w, remoteCTX)]
  }
}
"""
(1)COMPOSITION: domC extends domA:
=> '/a/b/x' would be covered by domA,
=> '/a/b/c/d' would be covered by domC.
(2)CONFLICT: domD conflicts with parent domain:
=> domA forbids remoteCTX from writing to '/a'.
"""
```

Listing 2: Policy resolution: composition vs. conflict

2.3 Transition rules and safety verification

In CMCAP, access control queries are formulated as reachability proofs in an information flow graph determined by the system's effective security policy.

Definition 6. *Given a CMCAP capability* $x = (p, c)$, *let* $perm(x) = p$ *be the permission encoded in* x *and* $ctx(x) = c$ *its operational context. For a pair of domains* d_1 *and* d_2, $flow_{d_1,d_2} = \{(c_1, c_2) \in capset_{d_1} \times capset_{d_2} \mid ctx(c_1) = ctx(c_2) \wedge perm(c_1) = \texttt{READ} \wedge perm(c_2) = \texttt{WRITE}\}$.

Definition 7. *Let* P *be the set of containment domains that constitute the effective policy. P's* `policy graph` *is a graph* $G_P = (V, E)$, *where* $V = \{(d, c) \mid d \in P \wedge c \in capset_d\}$ *and* $E = \{(u, v) \mid u = (d_1, c_1) \wedge v = (d_2, c_2) \wedge (c_1 = c_2 \vee flow_{d_1,d_2} \neq \emptyset)\}$. *Paths in* G_P *are called state transitions.*

For any given policy, a corresponding state transition graph expresses the system's invariant protection state. Verification for a safety property in the policy can thus be carried out by considering the policy's graph and proving the absence of paths that constitute a violation of the given property.

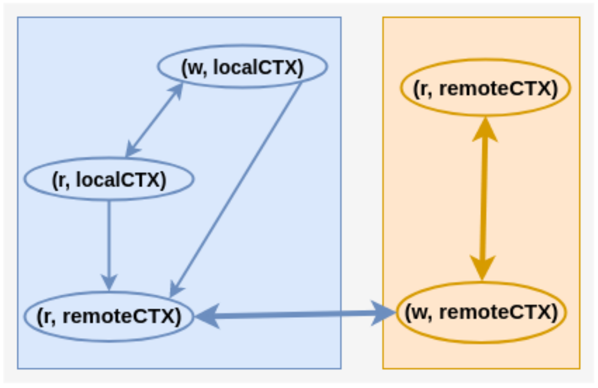
Definition 8. *A* `safety property` *expression is the negation of forbidden state transitions in a policy graph.*

Figure 1 illustrates a state transition graph representing a given CMCAP policy. Given a program in any state, its possible capabilities can be enumerated by computing all reachable nodes from its current state. Under the given policy, for instance, a program must transition into a state that has the `(r, localCTX)` capability in order to read files in the docsDMN domain, transition into the `(w, remoteCTX)` state to send data to the network, etc. The policy in Figure 1 therefore does not allow a program to write to files under "/home/alice/docs" after reading network data (here generically represented as "`network/**`").

Figure 1: Reference CMCAP policy (a) and corresponding state transition graph (b)

```
{
  docsDMN:{
     objects:["/home/alice/docs"],
     policy:[
        (r, localCTX), (w, localCTX),
        (r, remoteCTX)
     ]
  },
  networkDMN:{
     objects:["network/**"],
     policy:[
        (r, remoteCTX), (w, remoteCTX)
     ]
  }
}
```

(a) (Listing)-Reference CMCAP policy



(b) State transition graph

2.4 Ephemeral sandboxes

In cases where denying access to certain resources would cause the offending program to crash, CMCAP uses the virtual WRITE permission to divert the program's side-effects to a sandbox. Moving a program into a sandbox is better than terminating it when it is preferable to preserve the program's runtime correctness while respecting the system's policy. Sandboxes under CMCAP are ephemeral in that they are automatically generated when needed to provide virtualized resources to a program and may not persist beyond the program's execution context. A containment domain provides support for ephemeral sandboxes by allowing the 'w/#' (virtual WRITE) permission to files that ought to be virtualized for a given context.

3 ACCESS CONTROL SEMANTICS UNDER CMCAP

3.1 Permitted capabilities for a running program

Let *Dom* be the set of security domains, *Ctx* the set of operational contexts, and *Perm* the set of permissions. Moreover, let $exec\_dom\_ctx(p, d, c)$ characterise a state in which $c \in Ctx$ is attached to process p in domain $d \in Dom$.

Definition 9. *A* `context map` *for a protected object o is a condition expressed as follows:*

$$ctx\_map(D, C, R, o) \leftarrow$$
$$\exists(D, C, R) \in Dom \times Ctx \times Perm, o \in objset_D \wedge (R, C) \in capset_D.$$

Being a static constraint, an object's context map can be determined by enumeration in the effective policy.

Definition 10. *For a requested permission r, let p be the requesting process and o the target object.*

explicitly-permitted capability *:*
An `explicitly-permitted capability` *encodes a state characterised as follows:*

$$ctx\_cap(p, o, r) \leftarrow ctx\_map(D, C, r, o) \wedge exec\_dom\_ctx(p, D, C).$$

virtually-permitted capability *: If r is a virtual permission, the encoded state is a* `virtually-permitted capability`*, characterised as* $sandbox(p, o, r)$.

Process *p* acquires permission *r* on object *o* if *p* assumes a context that allows permission *r* in a domain that provides access to object *o*. The necessary precondition is expressed as

$$pre\_auth(p, o, r) \quad \leftarrow \quad ctx\_cap(p, o, r) \ \vee \ sandbox(p, o, r)$$

A permitted capability is therefore a state whose reachability depends on the current state and transition rules in the effective policy.

3.2 State transition schemas

State transitions in CMCAP are induced by actions that encode a binary relation between the current state and its successor state. Actions are state bindings of parameterized operators with the following schema :
$action(op(params), effects(op, params)) \leftarrow precond(op, params)$, `op(params)` being an instantiated action produced by binding parameters `params` to operator `op`.
The action's effects describe an atomic transition when the action succeeds. Its preconditions describe constraints that must be satisfied in the current state for the action to succeed.
Let `precond` be an action's preconditions, `effects` the list of fluents made true by the action, and `negated_effects` the list of fluents deleted by the action.

Definition 11. *If S_i is the current state and precond $\subset S_i$, the special operator* `apply` *induces a transition of the system into S_{i+1}, defined as follows:*

$$S_i \xrightarrow[apply(action)]{} (S_i \cup effects) \setminus negated\_effects$$

The `apply` operator is a no-op if $precond \not\subset S_i$.

Definition 12. *Given an initial state S_i and a goal state S_n, a plan is a sequence of transitions* $(S_{i+k} \xrightarrow[apply(action_k)]{} S_{i+k+1})_{0 \leq k \leq n-i}$.

The plan need not be unique. Goal strategies beyond proving the existence of "some" valid plan (for example heuristics for producing the best plan) are beyond this article's scope.

3.3 Access control queries

An access control question is formulated as a logic program, with the access request stated as a goal whose reachability translates into an approval. Specifically, the goal consists of the predicate $pre\_auth(p, o, r)$ in section 3.1.
To model access control decisions, CMCAP defines the following goals:
(1) $acq\_read(p, o) \leftarrow pre\_auth(p, o, cap\_READ)$, and
(2) $acq\_write(p, o) \leftarrow pre\_auth(p, o, cap\_WRITE)$.

4 CMCAP-LINUX: IMPLEMENTATION AND PRELIMINARY EVALUATION

4.1 Linux integration

CMCAP-Linux is an implementation of CMCAP based on the Linux Security Modules (LSM). The LSM framework [15] was chosen because many security mechanisms on Linux already use it to control operations on kernel objects. CMCAP-Linux can be configured as the default security module in the kernel build configuration, or enabled at boot time with the "`security=cmcap`" boot parameter. CMCAP-Linux is composed of two main parts: the LSM interface which is an observation point for system calls, and the CMCAP resolution engine which is responsible for tracking information flows and making access control decisions. A third component implements ephemeral sandboxes based on Linux namespaces.

4.1.1 Observation points. The LSM framework provides an interface for security modules to manage security fields on kernel objects and perform access control. CMCAP-Linux uses LSM hooks as observation points for system calls, which it translates into their corresponding information flow categories (READ, WRITE). For identifying file, pipe, and socket objects, CMCAP descriptors are attached to the security fields in both `struct file` and `struct inode` fields of the corresponding objects. The LSM provides a security field in `struct msg_msg` for IPC messages, and one in `struct msg_queue`'s perm (`struct kern_ipc_perm`) field for message queues. CMCAP descriptors will be attached to those fields in order to identify IPC objects. On program execution, a descriptor is attached to the program's `struct cred` field (inside `struct linux_binprm`). CMCAP descriptors are initialized with security contexts derived from the target objects' origin. For open files, the file's path is used to determine its containment domain according to the policy in effect. Socket and pipe objects are also identified the same way thanks to the sockfs and pipefs pseudo-filesystems. On program execution, the program's credentials are initialized with context from its source file, later adjusted with context from its interpreter (the program that called exec() to start the new program). Processes in the program inherit its credentials in their `struct task_struct`'s security field.

Access control decisions are made based on the requesting program's CMCAP descriptor, the target object's CMCAP descriptor, and the type of information flow derived from the requested operation. A list of operations mediated by CMCAP-Linux is presented in Table 1.

Table 1: Mediated system calls in CMCAP-Linux

Object	Operations
File	read, write, rename, symlink, sendfile, mmap, ioctl, fcntl, flock
Socket	create, read, write
Pipe, anonymous inode	create, read, write, splice, vmsplice
Shared memory segment	shmget, shmat, shmctl
IPC messages and queues	msgget, msgsnd, msgrcv, msgctl
Process	exec, ptrace, prctl

4.1.2 Ephemeral sandboxes. On Linux, namespaces constrain a process' view of the execution context, and their manipulation allows disassociating a process from the rest of its initial group. CMCAP uses the following namespaces when creating ephemeral sandboxes:

IPC namespace : to isolate the process' view of SysV IPC and POSIX messages queues;

Mount namespace : to hide changes to the file system, mounts, and open file descriptor tables;

UTS namespace : to hide changes to the hostname and NIS domain name.

The implemented isolation allows mapping virtual resources in a program's namespaces without affecting the rest of the system. Specifically, it prevents the sandboxed program from modifying the protected resources while at the same time letting the program function as it normally would. Resource emulation is implemented by making changes to a program's mount namespace to replace file system entries for the virtualized files by a context-private temporary file. The temporary file overlays the protected file and is only visible to programs in the same operational context. The overlay files are created using a pseudo-filesystem (cmcapfs) that can be configured to save written data to a memory backed store to provide consistent READs for the duration of the sandboxed program's execution context or simply divert file WRITEs to '/dev/null' if consistency is not required.

4.1.3 Dynamic application containment example. Based on the reference policy in Figure 1, the following runs of the same program will result in different runtime profiles:

(1) `ping -c 1 127.0.0.1 2>> /home/alice/docs/ping.log` : access denied.

(2) `ping -h 2>> /home/alice/docs/ping.log` : output written to ping.log.

In (1), the file's domain forbids network-originated content. Note that (2) succeeds because the "`ping -h`" command simply prints documentation, which does not involve network interaction.

4.2 Performance Considerations

A brief measurement of CMCAP's performance overhead was conducted, focusing on operations that involve creating or updating CMCAP security descriptors. The micro-benchmark in Figure 3 is divided into three categories, all run in tight loops: (1) open/-close operations to measure the overhead of the "open" system call, (2) writing a 1M file using a 1K buffer size to measure read-/write throughput, and (3) fork+execve operations to measure the overhead of starting new programs. The number of operations was varied per group for timing purposes: 10 million operations for category (1), 10,000 operations for (2) and (3). 10 iterations were averaged for each category.

For performance results in this section, CMCAP-Linux was built and tested on Linux 4.4.6 booted on an Intel Core2 Duo E8400 running at 3GHz with 8GB of RAM. The benchmark was run on 3 configurations: DAC, CMCAP-Linux, and AppArmor. AppArmor was run in complain mode, with policies generated using its profile generation utility (aa-genprof).

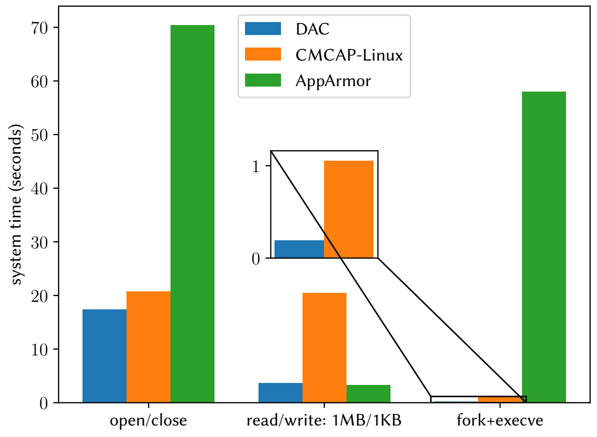


Figure 3: Resource access overhead: CMCAP_Linux vs. DAC vs. AppArmor

CMCAP's resource access mediation overhead is illustrated by the read/write results in Figure 3: while DAC and AppArmor make access control decisions once per resource (on open), CMCAP must update security descriptors following each information flow-related operation (read, write). In the remaining categories (1 and 3), CMCAP-Linux significantly outperforms AppArmor. The observed performance degradation for AppArmor is due in part to inefficient policies generated by its policy generation tool (the length of a profile generated by the tool grows with a program's execution path).

5 RELATED WORK

5.1 Modeling information flows with logic programs

Constraint logic programming is used in [1] to specify and reason about temporal RBAC policies. The expressiveness and flexibility of logic programming are a fundamental inspiration to CMCAP-Linux's userspace policy analysis tools.

A threat response system based on contextual security policies is described by Hervé Debar et al. [8]. A common characteristic between [8] and CMCAP is the application of contexts to dynamically model updates to security policies. Threat characterisation is done in [8] by analyzing known vulnerabilities and attack scenarios in order to define threat contexts for use in policy specification. In contrast, CMCAP policies assert permissions for defined contexts.

The authors in [3, 11, 12] use logic programs to model and analyze dynamic access control systems for verifying security properties and detecting explicit information-flow vulnerabilities. CMCAP is similar in that it models access control on containment domains which can be compared to protection boundaries across which flows are considered in [12]. Similarly, access control semantics in CMCAP is defined with an extended situation calculus interpreted as a logic program and query inference is used to prove reachability given an access control question expressed as a goal state. As an access control system, however, CMCAP is an implementation of the logic model as a policy enforcement point in the operating system.

5.2 Information flow tracking

The work in [6] uses a compiler-assisted static analysis to verify the reliability of information flow control (IFC) systems built on the LSM framework. The study highlights flaws in LSM-based IFC systems due to race conditions and incomplete tracking of indirect flows. A race condition-free implementation designed to address the highlighted flaws is demonstrated in RfBlare [5] by the authors. Similarly to previous IFC systems such as Blare [4], Laminar [13], and FlumeOS [10], the study in [5, 6] uses a stateful, temporal characterization of information flow-generating events to derive and validate resulting states and their compliance with a high-level information flow policy. Casting an access control decision as a logic program makes CMCAP implicitly stateful and temporal. For a precise information flow tracking, however, future implementations of CMCAP will require a framework adapted for that purpose.

6 CONCLUSION

Program-targeted mechanisms for fine-grained access control are inflexible. Their main usability challenges are that (1) understanding security properties for a single resource requires scanning more than one program policy, (2) specifying a fine-grained policy requires a high technical understanding of the targeted program, and (3) a long-term coordination is required to maintain consistency between programs and security policies. These challenges complicate security configuration and auditing, often resulting in common users relaxing security to run programs frictionlessly.

CMCAP proposes an adaptive mechanism for specifying and enforcing access control policies that dynamically reshape programs' capabilities based on their runtime profiles. It addresses the highlighted challenges by allowing users to specify policies for protected resources regardless of running programs. Virtual capabilities are provided to automatically sandbox offending programs when runtime compatibility is required.

Future implementations of CMCAP will focus on a more precise information tracking, and integration in more operating systems.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under awards 1241768 and 1637291.

REFERENCES

[1] Steve Barker and Peter J. Stuckey. 2003. Flexible Access Control Policy Specification with Constraint Logic Programming. *ACM Trans. Inf. Syst. Secur.* 6, 4 (Nov. 2003), 501–546. https://doi.org/10.1145/950191.950194

[2] Mick Bauer. 2006. Paranoid Penguin: An Introduction to Novell AppArmor. *Linux J.* 2006, 148 (Aug. 2006), 13–. http://dl.acm.org/citation.cfm?id=1149826.1149839

[3] Avik Chaudhuri, Prasad Naldurg, Sriram K. Rajamani, G. Ramalingam, and Lakshmisubrahmanyam Velaga. 2008. EON: Modeling and Analyzing Dynamic Access Control Systems with Logic Programs. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS '08)*. ACM, New York, NY, USA, 381–390. https://doi.org/10.1145/1455770.1455818

[4] Laurent George, Valérie Viet Triem Tong, and Ludovic Mé. 2009. Blare Tools: A Policy-Based Intrusion Detection System Automatically Set by the Security Policy. In *Recent Advances in Intrusion Detection*, Engin Kirda, Somesh Jha, and Davide Balzarotti (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 355–356.

[5] Laurent Georget, Mathieu Jaume, Guillaume Piolle, Frédéric Tronel, and Valérie Viet Triem Tong. 2017. Information Flow Tracking for Linux Handling Concurrent System Calls and Shared Memory. In *Software Engineering and Formal Methods*, Alessandro Cimatti and Marjan Sirjani (Eds.). Springer International Publishing, Cham, 1–16.

[6] Laurent Georget, Mathieu Jaume, Frédéric Tronel, Guillaume Piolle, and Valérie Viet Triem Tong. 2017. Verifying the Reliability of Operating System-Level Information Flow Control Systems in Linux. In *2017 IEEE/ACM 5th International FME Workshop on Formal Methods in Software Engineering (FormaliSE)*. 10–16. https://doi.org/10.1109/FormaliSE.2017.1

[7] William R. Harris, Somesh Jha, Thomas Reps, Jonathan Anderson, and Robert N. M. Watson. 2013. Declarative, Temporal, and Practical Programming with Capabilities. In *2013 IEEE Symposium on Security and Privacy*. 18–32. https://doi.org/10.1109/SP.2013.11

[8] Frédéric Cuppens Nora Cuppens-Boulahia Hervé Debar, Yohann Thomas. 2008. *Response: bridging the link between intrusion detection alerts and security policies*. Advances in Information Security, Vol. 38. Springer-Verlag, New York, NY.

[9] Boniface Hicks, Sandra Rueda, Luke St.Clair, Trent Jaeger, and Patrick McDaniel. 2010. A Logical Specification and Analysis for SELinux MLS Policy. *ACM Trans. Inf. Syst. Secur.* 13, 3, Article 26 (July 2010), 31 pages. https://doi.org/10.1145/1805974.1805982

[10] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. 2007. Information Flow Control for Standard OS Abstractions. *SIGOPS Oper. Syst. Rev.* 41, 6 (Oct. 2007), 321–334. https://doi.org/10.1145/1323293.1294293

[11] Prasad Naldurg and Raghavendra K.R. 2011. SEAL: A Logic Programming Framework for Specifying and Verifying Access Control Models. In *Proceedings of the 16th ACM Symposium on Access Control Models and Technologies (SACMAT '11)*. ACM, New York, NY, USA, 83–92. https://doi.org/10.1145/1998441.1998454

[12] Prasad Naldurg, Stefan Schwoon, Sriram Rajamani, and John Lambert. 2006. NETRA:: Seeing Through Access Control. In *Proceedings of the Fourth ACM Workshop on Formal Methods in Security (FMSE '06)*. ACM, New York, NY, USA, 55–66. https://doi.org/10.1145/1180337.1180343

[13] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. 2009. Laminar: Practical Fine-grained Decentralized Information Flow Control. *SIGPLAN Not.* 44, 6 (June 2009), 63–74. https://doi.org/10.1145/1543135.1542484

[14] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. 2010. Capsicum: practical capabilities for UNIX. In *Proceedings of the 19th USENIX Security Symposium*. http://www.cl.cam.ac.uk/research/security/capsicum/papers/2010usenix-security-capsicum-website.pdf

[15] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. 2002. Linux Security Modules: General Security Support for the Linux Kernel. In *Proceedings of the 11th USENIX Security Symposium*. USENIX Association, Berkeley, CA, USA, 17–31. http://dl.acm.org/citation.cfm?id=647253.720287