Cap9: A security protocol and framework for smart contracts

Jacob Payne Jake O'Shannessy Alexey Troitskiy

May 7, 2019

Abstract

We describe a secure and extensible operating system for smart contracts. Using a capability-based exokernel protocol, we can facilitate secure isolation, perform upgrades in a secure and robust manner, and prevent privilege escalation at any point in the development process. The protocol is intended to serve as an open standard and building block upon which more advanced and tailored security models can be built.

Contents

1 Introduction

> "Every program and every privileged user of the system should operate using the least amount of privilege necessary to complete the job."
>
> — Jerome Saltzer, Communications of the ACM

The underlying idea of decentralized organizations is the use of blockchain technology to securely manage and track a broad range of predominantly financial interactions without requiring a trusted third party.

While this unlocks a huge potential, the development and deployment of such organizations has a high barrier of entry. Smart contract[1] development practices and tooling are still in their prenatal stages of maturity, with zero margin for error. Additionally, organizations must be extensible upfront to allow changes and upgrades to their systems while still maintain high standards of safety.

Cap9 is a minimal open-source exokernel[2] protocol that serves as a building block for establishing secure and scalable organizations. Cap9 enables organizations that will grow in complexity and will require long term extensibility.

Using Cap9, users can establish an organization on the blockchain with a much higher level of confidence in its stability and security. The safety model Cap9 provides not only reduces risk, but allows organizations to be designed in a much more flexible and robust manner.

The exokernel model of Cap9 allows organizations to freely define their own systems and procedures, but within a safe and controlled environment. The Cap9 kernel provides the building blocks and primitives that organizations need to build more nuanced security and governance models to suit their use case. Most critically, by using the primitives Cap9 provides, designers of these systems will then be able to demonstrate these security guarantees to others without requiring manual-code verification.

2 Problems

Smart contracts are, by their nature, unforgiving machines that follow their specification to the letter. This is one of the main benefits of smart contracts, but it can also be their downfall, as the machine will faithfully execute any flaw or mistake in that code. This is a well known property of smart contracts, and is tackled in a variety of ways. One example is by employing lower level primitives like multi-signature wallets or escrow accounts. Using these tools we can encode a way in which participants in a transaction can "back out" of a transaction, or ensure that the correct approvals procedure is in place. The developers of smart

[1] A more appropriate, accurate name for smart contracts may be stored procedures (as proposed by Vlad Zamfir and used in database terminology). Smart contracts have also been critiqued by **?**.

[2] An exokernel exposes as much of the underlying hardware the system as possible, without larger abstractions such as filesystems. In Cap9 we expose all the underlying mechanisms (such as storage) directly to the contract, with only the permissions layer in-between.

contracts can thereby build a more flexible and forgiving system on top of the raw and unforgiving Ethereum machine underneath.

Particularly for financial operations, building higher level systems with these features is clearly necessary to be able to handle complex situations. However, these higher level systems dramatically increase the complexity of the system one is using. With this complexity comes another risk: one small mistake or vulnerability in the code can bring down the whole system. This forces users to strike a balance between flexibility of the system and complexity of the code. Increasingly, this balance has fallen more on the side of complexity.

2.1 Isolation

As smart contracts embrace this concept they are growing in size and capabilities. Complexity is progressively becoming a larger and larger issue. Already today we see many smart contracts being comprised of tens of different smart contracts operating together. These “smart contract systems” are also being provided with mechanisms to perform upgrades to their code, and the task of wrangling all of these issues is passed to the developer, with little support from their tools and underlying systems, and little oversight from their stakeholders.

Take an example of a smart contract system that contains multiple components. One component is tasked with maintaining the list of members of the group, and another is tasked with maintaining an allocation of tokens. As all of this code operates in the same space, there is nothing actively preventing the code which is supposed to be maintaining the list of members from modifying the allocation of tokens. The only thing preventing this code from reallocating all of the tokens, is that the developers and their auditors have correctly written and checked the code such that it doesn’t.

Developers and auditors are not weaponless in this fight, and rely on a variety of tools including DSLs (Domain Specific Languages), static analysis, and formal verification to write this code correctly and perform the right checks. These are excellent tools to combat these problems, however, they have two primary limitations:

- This is a difficult and complicated process. Much like any other complex process, a single mistake can result in a major flaw. If this is the single line of defence (as is common currently in smart contracts) then this flaw will not be caught by any other system. On a regular computer or server, if the code tries to do something too far out of its bounds it will be caught by the operating system. In smart contracts all code can make any change to the system, so an uncaught mistake has limitless consequences.
- These tools are available only to highly trained developers and auditors with the resources (i.e. time) to apply those skills to the code. Other stakeholders in the system are able to gain little or no insight into the code. They simply have to trust the developer.

2.2 Upgrades

One area where the lack of isolation in smart contracts is most problematic is in the upgrading of smart contracts. DSLs, verification, auditing, etc., are excellent tools to ensure that a smart contract will behave as intended, but where they struggle to maintain their effectiveness is during upgrades. Once an upgrade occurs, the whole system needs to be re-audited to ensure the same constraints still apply. With a multi-component smart contract, it is increasingly possible to update one or more of these components as the need arises, including for important security fixes.

Being able to change the code in a smart contract system is important, as it gives us the ability to correct errors. However, it also gives us the ability to introduce new ones or allow malicious actors to introduce them. In the example above, what was once a component to maintain a list of members can now become a component which siphons all of the funds held by the contract to an account held by a disgruntled developer.

In this example, the smart contract system suffered due to a lack of isolation. Perhaps it was a vulnerability in the members list component that was exploited, but the damage was far reaching and unrestricted. In traditional computing this is a common issue that needs guarding against, and is known as **privilege escalation**. This is also a concern in smart contracts without upgrades, but with upgrades the path to privilege escalation becomes easier, and more difficult to defend against.

3 Existing Work

The Ethereum smart contract blockchain is a concrete VM (virtual machine) which is very well defined and standardized. However, Ethereum itself provides no platform or protocols on which to base applications which have more complex security requirements (which is nearly all smart contract applications). A variety of platforms and protocols have been implemented and used which provide frameworks for authentication, voting, security checks and various other benefits. Many of these have been a success, but they are generally fairly specialised and fragmented. Each of these frameworks is built to serve a particular use case, and they are closely linked to the model of the organization they were designed for. This means that the code for providing security is closely intertwined with the application code.

The lack of a standard model for security mechanisms has so far contributed to many security vulnerabilities. One of the prominent examples was “The DAO” that was built to act as a hedge fund with a built-in proposal protocol to manage funds. The implementation was written in a highly specialized manner and did not separate the security mechanisms from the business logic. This lead to a large attack surface and ultimately a re-entrancy vulnerability that allowed the attacker to steal funds from the organization itself.

3.1 ZeppelinOS

Of the recent projects that are making an impact in this area is the ZeppelinOS[3] ZEP protocol by the Zeppelin Team. Users of smart contract libraries that use the ZEP protocol are able to vouch or vote for upgrades they wish to adopt. Only code that is sufficiently vouched for by the users is included in the system, with the developers working on that code being rewarded in cryptocurrency assets. The vouching system ensures that only code that has been approved by all the relevant users is included and allocates monetary rewards to incentivize high quality development work.

Unfortunately, this solution only solves a governance problem, which does not address the issue of limited introspection and transparency of the contracts themselves. In order to participate in this governance model, one would need to be able to understand and guarantee that every byte of code that one approves is correct. While the governance model gives one power over what code is included and what isn't, it doesn't equip one with the necessary tools or resources to make good decisions long-term.

Only highly skilled developers who are familiar with both blockchain contracts and the system involved have any hope of being able to understand the detail of code changes, and even they are limited in the resources they are able to spend on this activity. Regardless of what promises you are being told, unless you are a developer - the system is a black box. It requires sophisticated tools to be inspected and understood; without sufficient resources you cannot verify on your own how the system is actually implemented. Relying on developers alone - to verify these systems as we do today - will not be enough to legitimize smart contracts as secure financial instruments that the mainstream market can trust long-term.

3.2 AragonOS

AragonOS[4] is another example of smart contract framework that seeks to take a more operating system style approach. Much like Cap9, AragonOS features a permission system and a kernel, however, after that the design diverges significantly. AragonOS aims to be a featureful platform on which to build smart contract applications. While the permission system of Cap9 is targeted at isolating and limiting *code*, the permission system of AragonOS is targeted at a higher level, and is only one component of a vast suite of features of the AragonOS platform. We take a different approach in that the layer which isolates the various code components should be as separate and minimal as possible, and independent of the features built on top of it.

3.3 Domain Specific Languages

There is another branch of activity which has been able to make good progress in improving the state of the art, and that is by improving the languages and

[3] https://zeppelinos.org/

[4] https://aragon.org/

build tools of developers. This includes DSLs, static analysis tools, and even some that follow a similar ideal to Cap9 and incorporate a capability object model directly into the language. These tools are a great boon for developers and can dramatically reduce programmer errors. However, there is one very large drawback to these systems, which is that they are for the benefit of the developer. Once the code is built and running on-chain these tools have completed their task and are no longer in use. The responsibility is placed squarely on the developers to ensure that these tools are used correctly, and that the code that is running on chain accurately reflects the steps they took.

However, this still relies on completely trusting developers and that the constraints the developers believed they implemented were faithfully executed by an off-chain compiler. None of this information is included or verified *on-chain*.

To solve this, we propose a more general approach by creating an explicit security model for developing smart contracts that allows developers the ability to establish semantic restrictions that can communicate their intentions to their users. As an abstraction, restrictions that can be easily understood establish transparency and allow users an ability to verify these contracts on their own without relying on 3rd party trust.

4 Overview

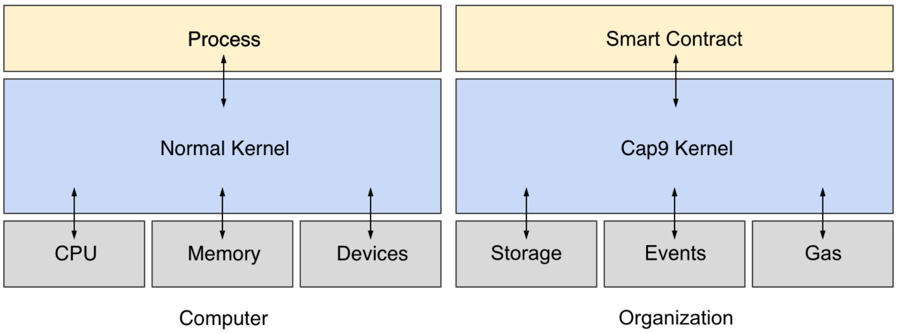

Figure 1: Similar to how existing operating systems act as an interface to hardware for applications on a computer, the Cap9 kernel provides an interface for smart contracts to securely access privileged assets and data of an organization.

In their early days, computers executed a sequence of instructions and manipulated input and output devices, blindly following the instructions laid out by the programmer. It was up to the programmer to ensure that the program they wrote was correct. Once the computer was executing the programmer had relinquished control and the program would execute as written.

Driven by a need to manage multiple programs, and eventually include restrictions, quotas, and permissions, operating systems were created. With operating systems, the programs weren't just executed blindly by the machine. When the machine encountered an instruction that required hardware access, it suspended the running program and passed that instruction to the operating system, which then decided what to do. This gave the operator of the machine control over what was and wasn't allowed to occur on the machine, at the cost of some run-time monitoring.

On Ethereum the situation is very similar to those early days of computers. Smart contracts are thoroughly audited and tested, but once they are on the blockchain they execute on the raw machine. If we want a system where we can interpose ourselves between our components and potentially dangerous resources (such as our storage) we need to make strong guarantees that components of our system will only abide by the policies we set, and won't break the rules.

In order to do so, we need to require that all contracts interact with critical resources only through system calls to our 'kernel' that we have placed in between the code and the dangerous resources. Once a contract is only operating through system calls, the operating system kernel has the final say on what the contract can or cannot do.

4.1 Procedures

A Procedure is a smart contract that can be executed by the kernel. This is the smallest independent component or unit of code. As far as the Ethereum VM is concerned they are a normal smart contract. Unlike normal smart contracts, procedures have several restrictions. Critically, procedures are only able to access state changes (i.e. the dangerous things) via system calls, and are not able to directly access storage, make outside calls or send transactions.

Much like processes are stored in a process table within a traditional operating system, procedures are stored within a Procedure Table held by the kernel. Each entry contains the id, capability list, and address of the procedure. Unlike a process table however, the procedure table does not maintain procedure state across separate transaction calls.

4.2 On-Chain Code Verification

In order to verify that a procedure follows certain restrictions we use On-Chain Code Verification. On-chain code verification is the use of a trusted smart contract to read and verify the code of an untrusted smart contract. By reading the code of an untrusted contract (via the EXTCODECOPY opcode), we can analyse it and make assertions about what it can and cannot do.

With on-chain code verification we can check if an untrusted contract has any opcodes that we don't allow. In particular this allows us to check if a contract can potentially: self destruct, make state changes, emit events, or make an external call. This might not seem like a big restriction, but the absence of such code allows us to make certain guarantees. For example: if the code does not contain the combination of SELFDESTRUCT, DELEGATECALL or CALLCODE, we now know that this code will never be able to be destroyed.

In our case, the trusted contract (which performs the verification) will be our kernel, while the untrusted contract would be a procedure. Before running a procedure, the kernel can check and verify the procedure, allowing the kernel to enforce certain restrictions or access control. The kernel only needs to do this check only once however, since in Ethereum the code of a contract can never change.

4.3 System Calls

As we covered above, a traditional operating system achieves introspection and control by interposing itself between the processes that are running and all other parts of the system including storage, memory, and even other processes. Each process is given its own memory and access to a processor, but is otherwise completely isolated. In order to do even the most fundamental task (be it print a character to the screen or read some stored data) it must ask the operating system to do so on its behalf. It is only through the operating system that the process can affect the real world. Via this mechanism, the kernel has complete control over what that process can and cannot do. These requests to the operating system that a process makes are called System Calls.

Cap9 also uses this system call mechanism, and the sequence of EVM opcodes that define a Cap9 system call are outlined below. The parameters of the system call are set prior to these 3 instructions, and are the responsibility of the designer of the contract (presumably with significant assistance from libraries).

Listing 1: Sequence of steps to perform a system call.

```
CALLER          // Get Caller
GAS             // Put all the available gas on the stack
DELEGATECALL    // Delegate Call to Caller
```

These instructions ensure that the contract is only calling to the original kernel instance and nothing more. Because this sequence of instructions is the only sequence of instructions with a state changing opcode (DELEGATECALL) we can use on-chain code verification to prove that a contract contains no state changing opcodes other than in the form of a secure system call.

The delegate call is a call back into the kernel. The kernel will only accept system calls from the procedure it is currently executing, ensuring the kernel has a complete understanding and control over what is being executed. The very first call the kernel makes (which is usually to the library code) is done via CALLCODE. This means that that our system, which we will call our "kernel instance", is the current storage and event space of the running code. It also means that the CALLER value, which is a global read-only value, is set to the address of our kernel instance. [5] When our procedure does a DELEGATECALL, this address is maintained. As a consequence, whenever a kernel is executing a system call, it is able to simply check that the CALLER value is equal to its own address (it is necessary for this value to be hardcoded into the instance, which is performed during instantiation).

[5] One downside of this design is that the original sender of the transaction must be included in the payload of the transaction if it is needed. As the kernel address is more security critical, and is needed more often, it is placed in the CALLER value.

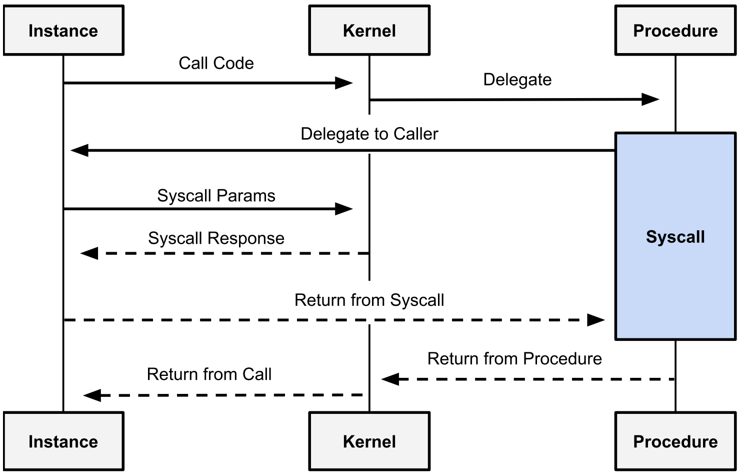

Figure 2: This diagram illustrates the general sequence of steps done when a procedure executes a system call

The process for executing a procedure is as follows:

1. Kernel instance does a CALLCODE[6](i.e. a regular library call) to the kernel library, executing the function in the kernel library that executes procedure.

2. The kernel library code uses the procedure table of the kernel instance to call the correct procedure using DELEGATECALL. During its execution, the procedure encounters a system call and performs a DELEGATECALL to the CALLER value, which is the kernel instance.

3. The kernel instance checks that itself is the original caller and if so, calls the kernel library for processing the system call.

4. After the system call is completed, the kernel library returns to the kernel instance, which then returns to the procedure.

4.4 Procedure Registration

Creating and adding a procedure to the kernel's procedure table (so that it can be executed) involves two steps. The first is to create and deploy the procedure as you would any other smart contract. Once it is deployed, it can then be registered with the kernel by simply giving the kernel the address of the code and letting the kernel handle the registration process. During this process, the kernel will first run the code through the verification process to ensure that the only state changing code it contains is run through the kernel. If the code passes this verification process, it is then added to the kernel's procedure table.

[6] The difference between CALLCODE and DELEGATECALL is that CALLCODE resets the CALLER value to whichever contract executed it. We use this to our advantage, as this sets the CALLER value to the kernel instance, allowing us to use it in system calls.

An important feature of this verification process is that it runs on-chain. Although more expensive then verifying things off-chain, this verification process is the core mechanism by which the kernel prevents execution of untrusted opcodes. This verification means that the 'proof' now lies on-chain, and we aren't simply trusting developers to have done the correct verifications.

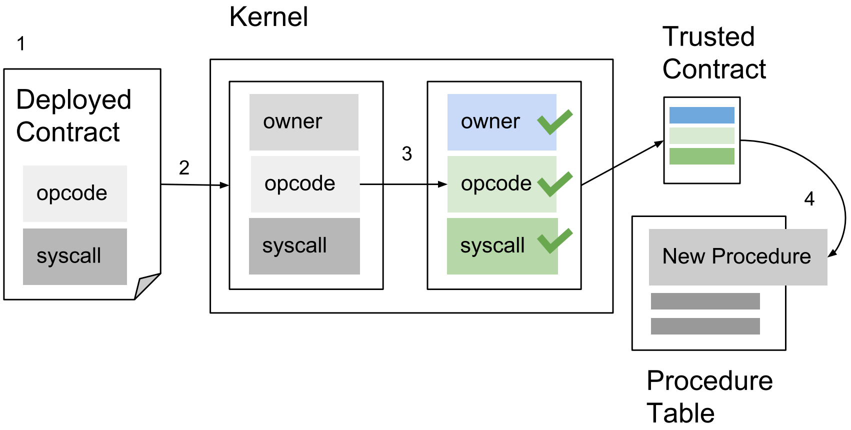

Figure 3: This diagram illustrates the general sequence of steps to verify and add protection to the procedure template for it to be included in the procedure table.

1. First, the contract is built and deployed as usual.
2. Second, the address of the contract is passed to the kernel for registration. The kernel parses the code of this contract and verifies it does not contain any illegal opcodes. If any illegal opcodes are found, the contract is rejected.
3. To designate a syscall dispatch, the code must include a valid syscall entry. To allow the kernel ownership, it checks if the contract has an additional header that prevents unauthorized calls to the contract.
4. Once the code as been verified it is added to the kernel's procedure table, and can then be assigned capabilities.
5. The procedure can then be used.

4.5 Entry Procedure

We have now outlined a kernel that is generally only accessible from its own procedures. It must, however, also accept some form of external transaction, which is the only way it can be triggered to execute code. As an operating system Cap9 should have no say over what kind of transactions and programs a system wants to execute, and should not impose any format on those transactions. Cap9 follows an exokernel design, where the kernel itself should stay out of the user's code as much as possible.

We can divide the calls that a Cap9 kernel has to accept into two categories:

1. A System Call, which will come from an executing procedure that the kernel itself called earlier.

2. An External Call from a 3rd party Ethereum address. This is what will trigger whatever actions the designer of the system has chosen to implement.

We have covered above how system calls are secured by ensuring that the kernel only accepts system calls from procedures it is currently calling. To allow the system to interact with users, we must also establish a way to handle external transactions.

In order to define an external interface to the kernel, we designate a procedure of our choice as the Entry Procedure. This entry procedure is created (and updated if need be) by the user, and is executed as a normal procedure. When the kernel instance receives an external transaction from another Ethereum address, it simply forwards the message data to the entry procedure. Thus whenever any transaction reaches the kernel, it is up to our entry procedure to decide what should happen and act as the interface to the kernel. This would look something like this:

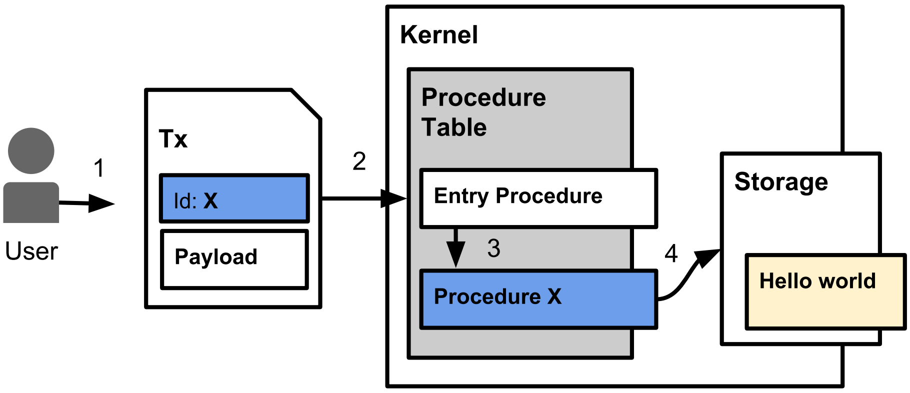

Figure 4: This diagram presents the general sequence of steps used for executing an entry procedure.

1. A user creates an order to execute a procedure. The format is defined by the entry procedure, but would usually contain the procedure id, and whatever payload the procedure needs.

2. The Kernel looks up the entry procedure in the procedure table, and dispatches it.

3. The entry procedure reads the payload and performs whatever user authentication logic the developer of the system desires.

4. If the payload is valid, the entry procedure looks up procedure X and calls it, procedure X does its primary function, which in this example is to store the string “Hello World” in storage. This is done using a syscall invoked using the necessary capability.

4.6 Auditability and the Principle of Least Privilege

Existing systems such as DSLs or libraries offer some level of verification and guarantees. Before the smart contracts are deployed, the developer is able to use these tools to verify the contracts. However, once these contracts are deployed they contain no information that can be used to readily verify the system. They have become opaque, and the verification information is only available to the developer.

By using an operating system model, Cap9 is able to enforce isolation at runtime. The information on the isolation of the system is contained within the kernel and can be audited quite simply by anyone looking at the system. This allows us to not only isolate the highest risk portion of our code and reduce our attack surface, but also to verifiably demonstrate that to others. To an outside auditor, many chunks of a Cap9 system will also be opaque black boxes. The difference is, this auditor will be able to see which permissions are applied to which black box. In other words the system becomes a collection of isolated and restricted black boxes.

Just as when running Linux or any other regular system, it is of course possible to run all code with full permissions (i.e. root). All of the system code can be included in a single procedure with full capabilities and the system will become as secure as a regular smart contract. But by applying isolation, Cap9 allows developers to separate their system into specialised components. These components that can then implement large amounts of logic without the risk of accidentally (or maliciously) executing some of the more dangerous functions of the system.

In order to do this, we need a security model to apply to procedures and system calls.

5 Security Model

We established two critical components of the Cap9 exokernel: introspection via system calls, and isolation via procedures to achieve the principle of least privilege.

System calls establish a way in which the operating system can now allow or deny various system calls. For this to be used effectively there needs to be a system for specifying policies. It is these policies that determine when the kernel should allow or deny a particular action.

In regular computing these restrictions do not generally exist within programs. It is not possible in most programming languages to import a library or module and say “my program sends packets over the network, and must have permission to do so, but this section of code coming from this library should not be able to”. This is the state of Ethereum security currently. Everything is built as a single program, without internal security boundaries available (the new STATICCALL opcode is an attempt to implement this).

However, now that we have established an operating system with system calls, we now have that point of control over the contracts running on our system, and we can craft policies to allow or deny any action that interacts with the rest of the system.

To implement these policies, Cap9 uses a capability-based access-control model. Access control governs all procedures; in order to perform a system call a procedure must possess an unforgeable token or key called a Capability. This capability identifies the particular resource a procedure wants to access, and simply by being in the procedures possession gives the procedure authority to use that resource.

5.1 Criteria for a Policy Mechanism

The tradeoffs in designing a policy mechanism are many. If the policy mechanism is too simple, it will not be able to provide the necessary security guarantees. If it is too complex, then the system becomes far less auditable and almost as complex as the procedures themselves.

Therefore, a policy mechanism must have two critical criteria:

1. It must be simple to read and analyze separately from the code it applies to.
2. It must be able to represent the guarantees about a system (i.e. the restrictions) that users wish to make.

These two factors are informed by the design already:

1. Given the high stakes and low quantity of code in smart contracts, this allows more complex policy mechanisms to be practical, as any system deployed as smart contracts must already be heavily audited.
2. Cap9 follows a microkernel approach so as to give as much freedom to the designer as possible. For this reason it is necessary to make as few assumptions about the use case of users as possible.

Amongst complex smart contract systems the exact requirements differ significantly, but the base requirements stay the same. The key goal is to improve the security and auditability of the system such that an external party or higher level “system designer” can be given control over what the various contracts in the system can do. This would allow them to compartmentalise areas of code and ensure that it only has the privileges it requires, allowing them to focus attention on more critical high-risk code. Even if another member of the organization updates a contract under his or her control, the system designer should be able to limit the potential damage of an error or malign action by sandboxing that contract.

The goal of the security model then becomes to design a system that can make these guarantees, and present it to those designing or auditing the system in such a way that they can be checked using fewer resources and less knowledge.

As an example: in the Linux kernel, the permission bit mask and owner on a user's personal file allows them to quickly and easily see (by listing the properties of their files with `ls -l`) that only they, and anybody with root access can access their files. The simple string presented by `ls -l` of `-rw-r----- bob users file.txt` instantly tells `bob` that (with the exception of root) only he can modify `file.txt`, while other users in the group users can also read it, but nobody else has access. This short simple string forms the policy placed on that file. It's simplicity means that the restrictions placed on that file are easily verifiable. It is not necessary to verify all the code on the system, because we know that the Linux kernel will enforce this policy.

5.2 Implementing a Capability Based Security Model

All policies in Cap9 are applied directly to procedures. Cap9 is a capability-based OS and therefore the policies take the form of a Capability List. Each capability in this list is an unforgeable key. This key is both an identifier for a resource, and an inherent permission to access that resource. The procedure cannot act outside this set of capabilities. Even if the procedure is updated, the limits of the procedure do not expand unless the capabilities associated with that procedure also expand.

By applying these capabilities at a procedure level, the policy mechanism becomes very simple. Where more complex policies are required, they can often be modelled by using simple procedures that "become" the policies and can be more simply audited.

It is critical to note that the capability system proposed here does not attempt to deal at all with "users". If a particular system includes users (which is to be expected) it is left to the creators of that system to dictate how that is organised and implemented.

For this reason, Cap9 aims to help system designers implement their own higher level permissions systems on solid primitives, rather than to anticipate all the use cases for those using Cap9.

Presented here is an outline of what a simplistic capability system might look like in Cap9. A procedure is created in two steps:

- Creation/Update: where the contract bytecode is uploaded to the kernel.
- Permission assignation: where somebody with the appropriate capability sets the capabilities of the new procedure.

When a procedure is created, it has zero capabilities available to it in its list. If, for example, it needs to modify the storage value at 0x7, it will need to be provided with that permission by a separate permission assignation. In this workflow, the procedure is deployed by a developer, and the permissions are assigned by the system designer once he approves this. The workflow around how permissions are requested and designed are left to the users.

The advantage of this design, where capabilities are assigned to procedures, is that it is simple and enforceable. Compared to some other capability system designs this does have a few limitations. One limitation is that it does not support dynamically chosen capabilities. For example this means that it is not possible to pass a capability of a particular storage location to a procedure. This is an often touted feature of capability systems that allows for the delegation of authority. The advantage of eschewing such a feature is that it makes the capabilities more static and auditable.

5.3 Custom User Permissions

A capability-based security model allows the flexibility to define a procedure that can act as a custom permission system akin to an access control list. With a custom permission system, users are free to define their own hierarchies or groups in order to satisfy their particular requirements. Thus users are not forced to follow a specific permission model that does not fit their needs and still maintain safety through establishing explicit access control within the system.

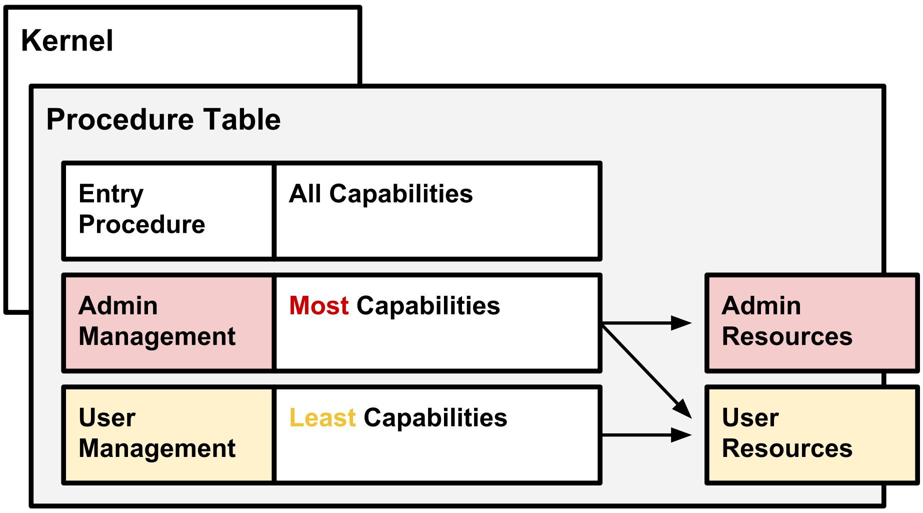

Figure 5: By separating admin management from user management, the principle of least authority can be applied.

An example access model can involve group decision making by vote, where a procedure is created to check votes and dispatch actions on resources within its allowed capabilities. Such a procedure can act as an administration system to maintain and upgrade previously defined components within the organization by secure vote.

Another example can involve groups, where a group is assigned control over a procedure with designated capabilities that define how the group can affect the organization. This in effect can be used as an explicit safeguard where each group cannot compromise the other's resources.

5.4 Kernel Objects

Table 1: Kernel objects.

Kernel Object	Capability Type	Description
Procedure	create	Create procedure with given identifier.
	push_cap	Add capability to procedure with given identifier
	delete_cap	Delete capability from procedure with given identifier and index
	call	Call procedure by given id and arguments.
	delete	Delete procedure by identifier.
	entry	Set the procedure with given identifier as the entry procedure.
Storage	read	Read from the memory by the given address.
	write	Write to the memory by the given address.
Log	write	Append log record with given topics.
Gas	received	The total amount of gas received from user.
	send	Send gas to an external address.

Each Kernel Object is a category of related capability types. Kernel objects designate the resources in the kernel that are protected by the kernel reference monitor and can only be indirectly accessed by invoking a system call.

5.4.1 Procedure Table

Sometimes, procedures need to access information about themselves as well as the state of the kernel. This includes the capabilities it has available as well as finding what other procedures are available. It is also important for procedures to able to dispatch other procedures if they are allowed to do so. This is usually when a procedure is used as a policy to check and verify if the appropriate signatures have been done. The procedure table object is a dynamic list of procedure identifiers that designate what procedures the kernel has available. Each procedure is defined by a unique identifier, contract address and a capability list.

5.4.2 Storage

Storage is the only way procedures can store and modify data. However, allowing procedures to directly access storage is unsafe and would inadvertently allow procedures to access or modify data that they do not own. To prevent this, storage is abstracted into capabilities that provide a separate key ranges for each procedure that they are allowed to modify. Additionally, procedures can provide access to other procedures to share or modify tables that they own.

The Storage object is defined by a 32 byte storage location where a procedure can either read or write a 32 byte value. In order to provide the kernel a protected storage space, storage is divided into two spaces, with half of locations assigned to kernel-space and and the other half assigned to user-space storage. This provides all procedures a user-space of 2^{255} unique keys. The number of storage keys far outweighs the capacity of the storage system itself, so will not be a limiting factor long term.

5.4.3 Events

Events are crucial for signalling verified changes to the outside world, accessing data asynchronously, as well as their use in establishing off-chain networks. In Ethereum, logs can be ascribed from 0–4 topics, with each topic being a 32-byte value. These topics are handled as addresses or namespaces. In order to log to a specific topic, the procedure must have the capability to do so. This capability is the write capability included under the Log object in the table kernel object table above. This capability type is then refined to dictate which topics it can write to.

5.4.4 Gas

The Gas object is defined as the total gas resources the kernel has available and designates how much resources are allocated to a procedure to spend during execution as well as how much gas it is allowed to send to an external address.

6 Summary

The Cap9 exokernel protocol provides a secure environment for organizations to establish extendible business logic through restricted smart contracts defined as procedures.

By using a low level capability-based security model, Cap9 allows developers to design custom permissions and policies that suit their use case, rather than lock organizations into a particular governance model. These will also form a standard set of primitives that can be used to reduce the burden of verification and communicate the security design of the system to others in a transparent manner.

7 Acknowledgments

We would like to express our gratitude to other projects notably our mentors and advisors, and the many welcoming people in the Ethereum community.