Evaluating SGX's Remote Attestation Security Through the Analysis of Copland Phrases

by

Freddy Eduardo Veloz Baez

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

April 2022

APPROVED:

Professor Daniel Dougherty, Thesis Advisor

Professor Craig Shue, Thesis Co-Advisor

Professor Craig Wills, Head of Department

Abstract

SGX is a set of extensions to Intel's chip architecture that allows a process to run securely in an isolated computing environment known as an enclave and store secrets that cannot be accessed by the system in which the process is located. This gives the necessary assurance to a remote party that the computations being run in the enclave can be trusted, which is a vital requirement in cloud computing. SGX achieve this via remote attestation, the process of receiving evidence over a network about the state of a target machine and appraising it.

In this work, we use Copland, a declarative language created to codify the intricacies of layered attestation protocols, to express SGX's attestation processes. We also analyze the different components that participate in the protocols and identify limitations that may be exploited, as well as other components that could potentially be future targets. We achieve these goals by analyzing the SGX Copland phrases with Chase, a model-finder tool presented by Rowe et al. that can find possible attack scenarios in an attestation protocol. By considering different assumptions and attack targets and producing seldom adversary scenarios, we are able to give insights into SGX's attestation security. Additionally, we explore some limitations in the way Copland can describe complex attestation protocols like SGX, the implications of these obstacles and how to respond to them when analyzing Copland phrases.

Contents

Chapter 1

Introduction

During the past decade, we have witnessed the rapid rise of cloud computing, which can be defined as a business information model in which computing resources located in datacenters are delivered as on-demand services over the Internet [2]. This rise has taken place thanks to cloud computing's unique characteristics, such as seamless hardware scaling based on the application's needs and the convenience of paying only for the computing resources required[11]. However, this kind of resource outsourcing comes with a unique set of challenges, one of which is the so called remote computation problem, which is defined by Costan et al. [5] as the difficulty of securely "executing software on a remote computer owned and maintained by an untrusted party", or more concisely expressed with the question: How can we trust with our secrets a remote system that might have been compromised without our knowledge?

Multiple solutions have been put forward to guarantee integrity and confidentiality in these kinds of scenarios. A few years ago, Intel released a set of extensions to its CPU architecture known as SGX, or Software Guard Extensions, which allows a process to create an isolated computing environment that cannot be accessed by

its surrounding system. These isolated environments, known as enclaves, can communicate securely with remote entities, which can verify the integrity of the enclave by assessing evidence of its launching state [1]. The process of receiving evidence over a network about the state of a target machine or an application and appraising it in order to determine if the system has been compromised or if it is working as intended is known as remote attestation [4].

Nevertheless, remote attestation processes can become increasingly complex as more entities get involved in the process. Finding all the scenarios in which an adversary can successfully attack an attestation process becomes prohibitive when working with complex attestations. In this context, an adversary is considered successful if they can influence the attestation process (whether it is the processes behind the measurements or the evidence itself being produced) in a way such that the appraiser concludes that the target system or application has not been compromised when one of them has. In the case of SGX's remote attestation, the process involves at least the following parties (present in three different locations): the service provider, Intel's Attestation Service, the quoting enclave, the application and the application's enclave [27].

Therefore, an attestation process like the one involved in SGX presents an attack surface of considerable size that requires a rigorous design phase to successfully prevent attack scenarios from being computationally feasible. Although Intel has kept some key aspects of SGX's design confidential, a number of vulnerabilities have been detected in SGX in the past few years [5] [28] [3]. The most serious one was described by van Schaik et al. and was named SGAxe [25]. This attack allows a malicious agent to steal part of the enclave's memory, which can include the EPID's attestation key, and therefore allow the attacker to pass malicious enclaves as genuine to Intel's Attestation Server. The repercussions of SGAxe are critical as it can help

bypass all security guarantees provided by SGX [25]. As SGAxe's inventors proved, these limitations are rooted in a design phase in which Intel adopted stronger security assumptions than what its hardware implementation could actually deliver, which raises the need to perfect these processes.

In order to improve a remote attestation design phase to make these issues easier to detect early in the process, different approaches can be followed, among which is formal analysis.

Copland is a declarative language presented by Ramsdell et al. [16] It can express layered attestation protocols through a syntax that abstracts the specifics requirements of these protocols and processes them as parameters of some basic actions that take place in all attestation processes: measurements and evidence bundling. Attestation protocols in Copland are expressed as "phrases". Each phrase consist of atomic actions that characterize different kinds of measurements that produce evidence. Other works [15] have modeled attestation protocols in Copland in order to better analyze the ways in which an attacker might compromise the system and go undetected with the use of a model-finding tool that works with Copland phrases [20]. We believe that a tool like Copland can be helpful at analyzing SGX.

Specifically, we seek to find out what a declarative language like Copland can tell us about the limitations and strengths of security related tools like Intel's SGX. Can it make it easier to recognize weak points in any of the components involved in SGX's attestation process? Can it help us identify the assumptions Intel likely made at the design phase regarding SGX's security? And if it does, have these assumptions been proven correct by other authors or can they be considered to not have been exhaustive enough?

Regarding Copland, we are interested in exploring whether the language is rich enough to fully express the nuances of complex attestation protocols like SGX. In

case there are limitations, to what issues can they lead? And how can we respond to them?

To answer these questions, we use Copland phrases to model the SGX attestation processes in order to gain insight into the strengths and limitations of its design. We also determine how they can lead to security violations that could jeopardize the trust in systems that rely on SGX. We ran a model-finder tool presented by Rowe et al. [20] called Chase on the Copland phrases and specified the attacker's goal to be able to identify the different attack models to which SGX's attestation processes might be vulnerable to. After that, we tested out a number of different assumptions, checked how the model reacts to them and determined which security assumptions were likely made by Intel when designing SGX (for instance, that some attack models would be computationally infeasible due to SGX's inherent characteristics) and whether these assumptions covered all possible attack models. We believe this work is a tangible example of the benefits of rethinking attestation protocols in terms of their abstract operations with languages like Copland during their design phase. Besides that, we believe it provides a security analysis of SGX from a new perspective.

Chapter 2

Background and related work

Some of the tools and concepts relevant to this work are: Intel's Software Guard Extensions (SGX), remote attestation and Copland. We also believe it is important to discuss previous research regarding SGX's vulnerabilities, to get an idea of some of the attack approaches that have been pursued against the technology. In the following sections, we provide some general information about these topics.

2.1 Software Guard Extensions (SGX)

Intel's Software Guard Extensions, better known as SGX, is a set of extensions to Intel's CPU architecture that allow a system to run software securely and store data that cannot be accessed even if the system itself is compromised [1]. In other words, it provides integrity and confidentiality to computations that run in a system that may be malicious [5]. This is achieved with the use of enclaves, which are software containers that are protected by hardware enforced policies. In this context, the system's trusted computing base consists of the software in the enclave, the CPU's firmware and its hardware [1]. In order to prevent the possibility of the enclave's code being manipulated or accessed, the processor isolates these computations from

the operative system, the hypervisor, and the rest of its environment [5].

Protection schemes like SGX aim to solve the secure remote computation problem, as it is imperative for service providers that communicate with clients over the Internet to find a way to confirm that the secrets they provide to the client are going to be secure no matter the state of the system in which it is running. To ensure this goal, SGX provides the following guarantees [1]:

- An enclave can request a secure assertion of its identity to the platform, which can be done with the EREPORT hardware instruction.
- An enclave can verify an assertion from another enclave on the same platform.
- A remote entity can verify an assertion from an enclave.
- An enclave can access keys that are bound to the platform, which is done with the EGETKEY hardware instruction.

A key concept when discussing SGX are the assertions of an enclave's identity. They allow other enclaves or an external entity, like a service provider or Intel itself, to ensure the integrity and the identity of the enclave. They are represented with the measurement register known as EREPORT. This report is a SHA-256 digest of the state of the enclave at the moment it is built, including, among other values, its code, data stored, stack and heap [1].

On the SGX threat model, we assume that a hypothetical attacker has control over the OS and can read any traffic between components of the device or externally. The only part of the device that can be trusted is the processor and no side-channel attacks are feasible. Intel is also a trusted party, which means that the private keys created during production are assumed to be secure [24].

2.1.1 Attestation in SGX

There are two kinds of attestation processes in SGX: local attestation and remote attestation. In local attestation, an enclave tries to prove its identity to another one that resides in the same platform. To achieve this, it produces a local quote of its identity by calling the EREPORT hardware instruction, which generates a report that includes a MAC tag that binds a message supplied by the challenging enclave and a protected derivation key that can only be accessed by genuine enclaves. The MAC is used to prove the authenticity of the target's enclave Record, which is part of the report and includes information about its identity, state and other attributes [7]. The MAC tag is a CMAC (block-cipher based MAC) that uses AES 128-bit as its encryption algorithm [5].

To verify this report, the challenging enclave has to call the EGETKEY hardware instruction, which derives a report key that can be used to recompute the MAC and compare it to the one sent by the challenged enclave. As EGETKEY takes as an argument the value that was binded in the report produced by the challenged enclave, a malicious enclave cannot impersonate another one [7]. For a more detailed explanation of SGX's local attestation process and the inner functioning of the EREPORT and EGETKEY instructions, check the explanation provided by Costan et al. [5]

In a remote attestation, a service provider that is not in the same platform as the enclave challenges it in order to verify its identity. The challenged enclave must send a remote quote, which is generated by a special enclave known as the quoting enclave, present in every SGX processor. The quoting enclave verifies the identity of the challenged enclave through local attestation and then signs the report by following Intel's EPID (Enhanced Privacy Identification) protocol and using the EPID keys that are obtained in a previously executed provisioning process with Intel.

The signed report is sent to the service provider and then to Intel's attestation server to verify its authenticity. As EPID follows a group signature scheme, Intel cannot track the identity of the processor which generated the quote, thus providing privacy and verifying the enclave's validity at the same time [7].

Another important concept is mutual attestation, which refers to the situation in which two entities need to attest to each other before they can establish a level of trust. Depending on the protocol's requirements, a mutual attestation can be performed in sequence or parallel. Mutual attestation in the context of Copland is explored in Chapter 4 [8].

2.1.2 Attacks against SGX

SGX is not completely secure. Possible attacks have been identified by authors like Costan et al. [5], particularly side-channel approaches that can leak useful information from the enclave to the attacker. An adversary that controls the machine's kernel can, for instance, learn the memory access patterns used by an enclave, as SGX's enclaves use the same address translation process of Intel architecture in which specific bits are set depending on the type of memory access. This kind of attack is known as a passive address translation attack. SGX is also vulnerable to cache timing attacks, another kind of side-channel attack. Subsequent research by Wang et al. [28] and Brasser et al. [3] have confirmed these vulnerabilities in SGX and shown that they can be used in practical attacks.

However, probably the most serious attack against SGX is SGAxe, first described by van Schaik et al. [25] and which has the potential to allow an attacker to impersonate enclaves. SGAxe is based on CacheOut, a Cache attack that exploits Intel's memory overwriting process and exfiltrates information via cache evictions [26]. At a high level, the attacker waits until the enclave reads data from the LD1 cache and

then injects its own data into this cache so that its target is evicted from it and sent to higher-level caches or to memory. When the enclave tries to read this data again, it is forced to look for it elsewhere and a cache miss occurs, which makes the processor store the data as soon as it is found in a line fill buffer (LFB). This data can then be extracted by the attacker via a TSX asynchronous abort (TAA) attack.

In SGAxe, a CacheOut attack is performed in order to extract the processor's AES sealing key from a LFB, which is then used to decrypt the EPID keys. An attacker can then instantiate a malicious quoting enclave that can then sign as valid any false attestation quote from an enclave under the attacker's control. As this report is signed using the correct EPID keys, Intel's Attestation Server will recognize it as valid and the service provider will start sharing secrets with it, therefore breaking the confidentiality and integrity of the whole system [25].

2.2 Copland

Copland is a declarative language presented in 2019 by Ramsdell et al. [16]: that seeks to give a unified syntax to express layered attestation protocols. In this context, a layered attestation is a kind of attestation process in which trust in a system is built by layers. In other words, different parts of a system are measured in a specific order layer by layer to progressively increase the level of trust in its integrity . Copland was developed with the intent of fulfilling three basic requirements

- Flexibility, as attestation protocols can differ widely with each other. To tackle this, Copland abstracts the similarities that these protocols share into measurements and bundling of generated evidence and sets the specific protocol-unique operations as parameters.

- Unambiguous execution, which is achieved by Copland's semantics, including

evidence flow and measurement order.

- Ability to be statically analyzed, for which Copland provides denotational semantics.

2.2.1 Copland syntax

A typical Copland phrase is defined by a place and a series of internal phrases, each of which is formed by atomic actions that interact with each other through Copland's grammar [13]. Each internal phrase can have its own location, or otherwise it is assumed that the actions it describes take place in the same location as its parent phrase. We can define this general syntax in the following way:

$$*\mathsf{PLACE} : \mathsf{PHRASE} \ldots \mathsf{PHRASE}$$

The place where a remote measurement takes place is set with the "@" symbol with the exception of the location of the first "parent" phrase, which is set with a "*" symbol. Places can be named with digits or uppercase characters. If no place is defined in a phrase nor its parent, it is assumed that the phrase takes place at location 0. The @ symbol has the highest precedence in a Copland phrase and can be nested, which is used in cases when a component needs to request a measurement be done by another one. As an example, if we wanted location p to request location q to execute a specific measurement, we would express it with the following syntax:

$$@_{\mathsf{p}}\,[@_{\mathsf{q}}\,[\mathsf{PHRASE}]]$$

A typical Copland atomic measurement is defined by an Attestation Service Provider (ASP), which requires four parameters: the type of measurement (m), the parameters of the measurement ($\bar{a}$), the location where it takes place (p) and its

target (r). For instance, if we want to run an antivirus on a file located at at place q with no additional parameters, we can express that in the following Copland phrase (notice that we indicate that the attestation is started by location q itself) [14]:

$$*\mathsf{q} : \mathsf{ASP\ av\ a\ q\ file}$$

Besides measurements defined by ASP, Copland has a number of primitives to express some common actions performed during attestations, like CPY, SIG and HSH, which refer to copying, signing and hashing, respectively [13]. The exact implementations of these primitives are defined in each protocol implementation based on its specific needs and not by Copland itself.

Copland phrases interact with each other via branching and linear sequencing, which define the order in which measurements of an attestation take place. Linear sequencing execution is denoted by the $\rightarrow$ character and requires a measurement to be performed strictly after the previous one, whose evidence is consumed by the later. In this context, consuming evidence refers to said evidence being used as an initial parameter of the next one. For instance, a Copland phrase might require a piece of software to be measured in some way and this measurement might be signed afterwards. In this example, the evidence produced in the first measurement is consumed by the second action, namely the signing of said evidence:

$$@_{\mathsf{p}}\,[\mathsf{ASP\ m\ a\ m\ r}\ \rightarrow\ \mathsf{SGN}]$$

Branching can be sequential or parallel and is used to split evidence to two phrases via a π branching function. Sequential branching is expressed with the symbol $<$ and parallel branching with the symbol $\sim$. At a high level, branching allows the evidence previously generated in the protocol to be shared among the

two phrases at both ends of the branching symbol. The way this evidence is shared is defined by the π function, which can state that both phrases get a copy of the evidence or that only one of them does. We can express this way of sharing evidence with the + or - symbols. So, if we want both phrases to get a copy of the evidence and we want the first phrase to be executed before the second one, we can express it in the following way:

$$\mathsf{PHRASE} +< + \mathsf{PHRASE}$$

On the other hand, if we want the evidence to be received only by the second phrase and we do not care which phrase is executed first, we can express it like this:

$$\mathsf{PHRASE} -\sim + \mathsf{PHRASE}$$

After both branches have been executed, the evidence produced by both is bundled together.

An example first described in [14] can be a good way of bringing these concepts together. Imagine that we want a virus checker vc to run an analysis of an application t located in p. In order to guarantee that the results we get are fresh, it is a good idea to send an initial nonce n at the beginning of the attestation process and have the result of the vc measurement be signed together with the nonce before returning the evidence. One additional level of security can be achieved by running a measurement h on the virus checker's infrastructure v to guarantee that the checker itself has not been compromised by an attacker. This measurement would be performed from an isolated domain ma that has access to p's environment. The Copland phrase for this attestation protocol would be:

@p n [@ma n [(ASP h ā p v) → SIG] → (ASP vc ā p t) → SIG]

2.2.2 Copland Analysis with Chase

In the paper "Automated Trust Analysis of Copland Specifications for Layered Attestations" [20], Rowe et. al explain the process to analyze Copland phrases via Chase and include a number of examples that grow in complexity in order to show its capabilities. We present one of them to give an idea of the kind of information we can get from Chase.

Imagine a simple bank website in which users can authenticate and do transactions. The bank asks for a username and password in order to authorize access to the user. However, in order to guarantee that the user's browser has not been compromised, let us say, by a malicious extension that could hijack the user's session and send unintended transactions, the bank asks the user to complete an attestation process in which it receives some evidence to verify if the user's browser is working as intended. This process could be done by a browser monitor bmon that would work from the userspace us outside the browser (so a malicious extension could not corrupt it) and check every installed extension exts to verify their states. Additionally, we can have an antivirus av that runs from the kernelspace ks that does a scan of the browser monitor itself in order to confirm it has not been compromised. This protocol can be expressed by the following Copland phrase:

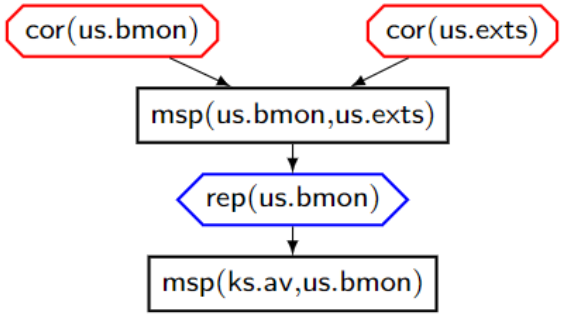
```
* bank : @ks [av us bmon] +~+ @us [bmon us exts]
```

In order to run our Chase analysis, we need to select a measurement of interest defined by the phi predicate. This measurement refers to the moment in time the attacker could be considered successful by having corrupted the desired component without being detected. In our example, the measurement of interest is the one

bmon does on exts, while exts is the component the attacker targets.

Running Chase with the provided inputs produces five different successful attack scenarios. Figure 2.1 shows one of them, in which an attacker corrupts both the browser monitor and the browser extensions. It then runs the measurement of interest, which would erroneously tell the bank that the extensions are working as intended. Finally, before the measurement of the browser monitor, the attacker repairs it in order to fool the antivirus.

Figure 2.1: Attack scenario on the bank attestation protocol.



The security of this attestation scheme can be improved by making some small changes. For instance, if we specify that the antivirus measurement has to be done strictly before the browser monitor measurement, the attack scenario presented in Figure 2.1 would not be possible. However, the reason to include this simple example was to give an idea about the attack scenario analysis done by Chase.

2.3 Related work

There has been significant work on remote attestation over the past decade. The main tenets of the process were established in 2011 by Coker et al. [4] They defined five principles that should serve as a base of any remote attestation protocol:

1. Evidence generated in an attestation must be fresh.

2. Measurements must be able to get all the required information of their target.

3. Targets should be able to decide which measurements are sent to an appraiser.

4. Attestations should have well-defined semantics.

5. Attestations should be able to prove their own trustworthiness.

A common approach when building trust in a system via an attestation is to perform measurements from deeper to upper layers in order to build trust "bottom-up", in a process that Rowe named layered attestations in a 2016 work [19]. In [18], Rowe demonstrates the validity of this approach and concludes that if a system performs a layered attestation, an adversary would need to either corrupt a deep component of the system or corrupt a component in a very short window of time, both of which are much harder tasks to accomplish. How the evidence is bundled in a layered attestation is of particular importance and Rowe proved in a subsequent work that common approaches, like storing evidence in a Trusted Platform Module (TPM), might not be secure enough against dynamic corruptions [17].

In order to model the complexities of these protocols, Ramsdell et al. [16] presented Copland, a language built with the main goal of describing layered attestations protocols while facilitating their formal analysis. Over the past few years, Copland has received additional support, like the creation of a Copland interpreter and attestation manager written in Haskell by Petz et al. [13], or the codification by Gray of TPM functions into the language in order to provide a root of trust for the different claims made during attestation protocols [6]. Helble et al. [8], for their part, addressed how to use Copland to model more complex attestation protocols that involved mutual appraisals, caching evidence and numerous principals.

The analysis of Copland phrases in order to find approaches an adversary might take to corrupt attestation protocols has been explored by Rowe [20], who in 2021

presented the Chase model-finder, which can be used to find different execution scenarios and attack models for a Copland phrase based on a set of initial assumptions, and which is of particular interest for this work as it was used to analyze the SGX attestation processes. Chase was based on work done by Saghafi et al. in their Razor model-finder [21], a tool presented in 2015 to analyze first-order theories and show them in geometric form.

An example of a protocol implemented in Copland and analyzed with the Chase model-finder can be found in [15], where the authors express the attestation process of a UAV flying system in Copland and refine the produced phrase by looking at the possible attack scenarios that the protocol is vulnerable to.

The pertinence of applying formal analysis of the security of SGX via its attestations protocols has been explored by Guttman et al. [7], which took SGX as an example case for analyzing hardware, trust and attestation rules in these kinds of protocols. Sardar et al. [22], on the other hand, used the ProVerif protocol verifier to confirm the security properties of third-party remote attestations based on SGX's attestation primitives, assuming the SGX threat model is correct.

In addition to SGX, there has been other proposals to build hardware-based trust in a system. A popular alternative to SGX is AMD Memory Encryption. Mofrad et al. [12] analyzed the two technologies and concluded that although AMD Memory Encryption offered faster performance for applications that need to use a lot of secure memory, it did not offer the same level of memory protections as SGX. Another approach is described in [9], where Jurgensen et al. propose using a seL4 microkernel that runs an embedded Linux virtual machine. This Linux implementation can run measurements on itself, but not on the seL4 layer, while the seL4 layer can measure both itself and the Linux system and ensure memory isolation.

Chapter 3

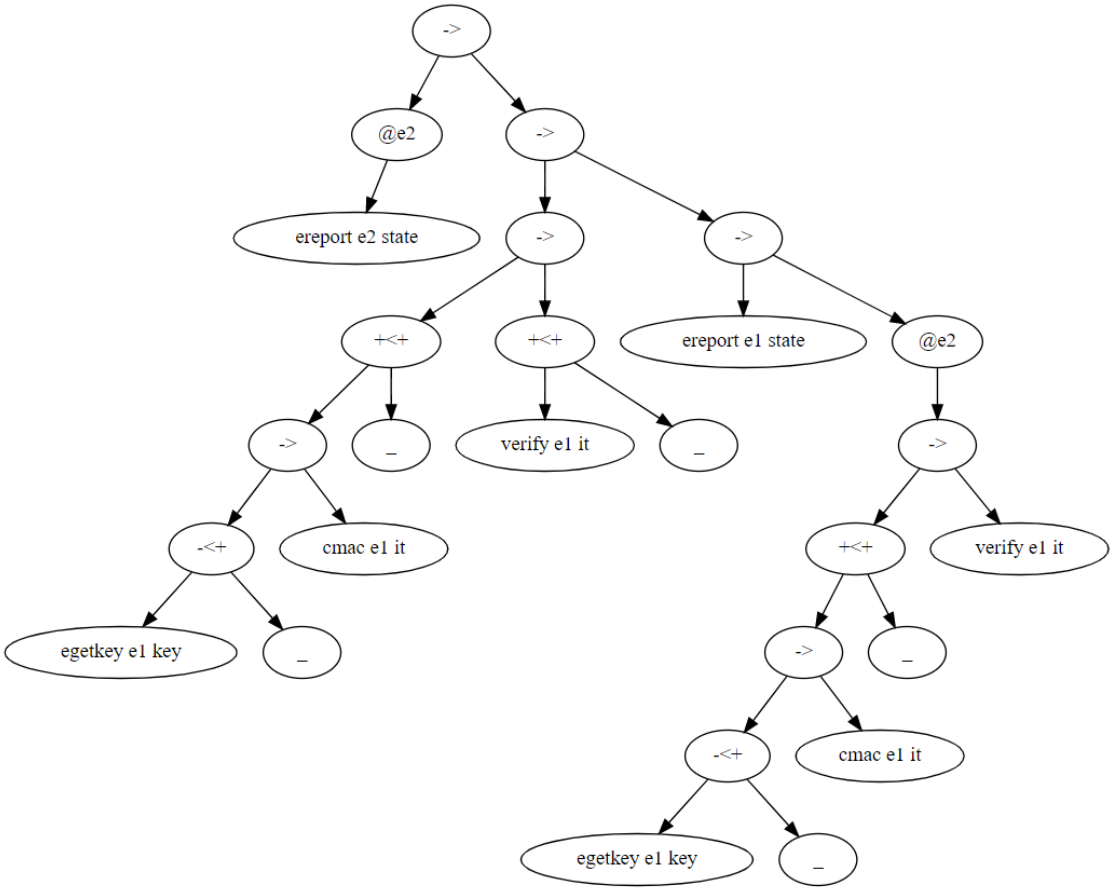
Approach

Our approach aims to detect possible limitations in the design of SGX's attestation protocol that could lead to confidentiality and/or integrity violations. We believe that Copland can be a valuable too to obtain the abstraction levels required to make the process manageable for complex protocols like the one used in SGX. We followed the template to analyze attestation protocols presented by Rowe et al. [20] The first step was to model SGX local and remote attestation processes as Copland phrases and analyze their denotational semantics and the events involved in the process.

We defined the operations involved in SGX as Copland actions. Some of them are:

- ereport, which refers to the SGX physical instruction of the same name.
- egetkey, which refers to the SGX physical instruction of the same name.
- verify, which the produced attestation quotes with the CMAC (block-cipher based MAC) obtained by egetkey.
- checkprivrl, which verifies if a processor's credentials are in Intel's revocation list.

Figure 3.1: Copland syntax tree for SGX's local attestation



This step allowed us to notice some limitations in Copland's syntax when expressing SGX's attestation protocols, which we explain in Chapter 4.

The following step involved analyzing our Copland phrases with a model-finder tool called Chase developed by the MITRE Corporation [20] to find the different attack scenarios to which SGX's attestation processes might be vulnerable. We then analyzed each attack scenario, classified them according to the level of damage that they could generate in practical attestation scenarios and identified which of them have been found in real life attacks against SGX.

We subsequently tried out different security assumptions, each of which reduced the number of attack models, for the different components in SGX's attestation

processes. We analyzed each scenario until we reached the assumptions that only left the attack models contemplated in the SGX threat model. By doing this, we were be able to get a glimpse of the considerations contemplated during Intel's attestation design phase, which could lead to a better understanding of many of the decisions that are still not well understood among researchers analyzing SGX [23]. We also weighed these assumptions against the guarantees provided by SGX's implementation [5] [25] [23].

As part of our conclusions, we found most of these assumptions to be based on SGX's inherent characteristics (isolation, hardware-enforced protection, etc), which ruled out most attack scenarios. Although adversary approaches that rely on deep or recent attacks usually represent an acceptable level of risk in attestation scenarios due to their complexity [18], Intel's security assurances regarding SGX and their described threat model should also rule them out. Our results regarding SGX are detailed in Chapter 5.

Thus, we hope that this work will provide valuable insights about SGX's security in a novel perspective and show the benefits of using a declarative language like Copland in the design phase of any protocol that requires some kind of remote attestation.

Chapter 4

Limitations of Copland

In this chapter, we explore some of the difficulties we found while modeling the SGX attestation protocols with Copland, particularly in relation with mutual attestation. Some of the approaches to address them are discussed in this chapter and then implemented in Chapter 5.

4.1 Mutual attestation and the "appraise" action

SGX's local attestation, which is a part of their larger remote attestation process, is an example of mutual attestation, as it requires two enclaves located in the same machine to attest their identity to each other. Only after both enclaves have proven they are running in the same machine and that they have not been compromised, they can trust each other and start exchanging information or running any other operation that requires a certain level of trust between them.

Although there are not many mentions of mutual attestation protocols in Copland's documentation, Helble et al. [8] include one such example of a protocol that includes two principals, P0 and P1, that attest their identity to each other and then send the evidence generated in their respective measurements to a third component

P2 that appraises it. This information flow can be seen in Figure 4.1. As the protocol does not require a specific order in which the attestations have to take place, this example can be expressed with two different Copland phrases that describe attestations running in parallel:

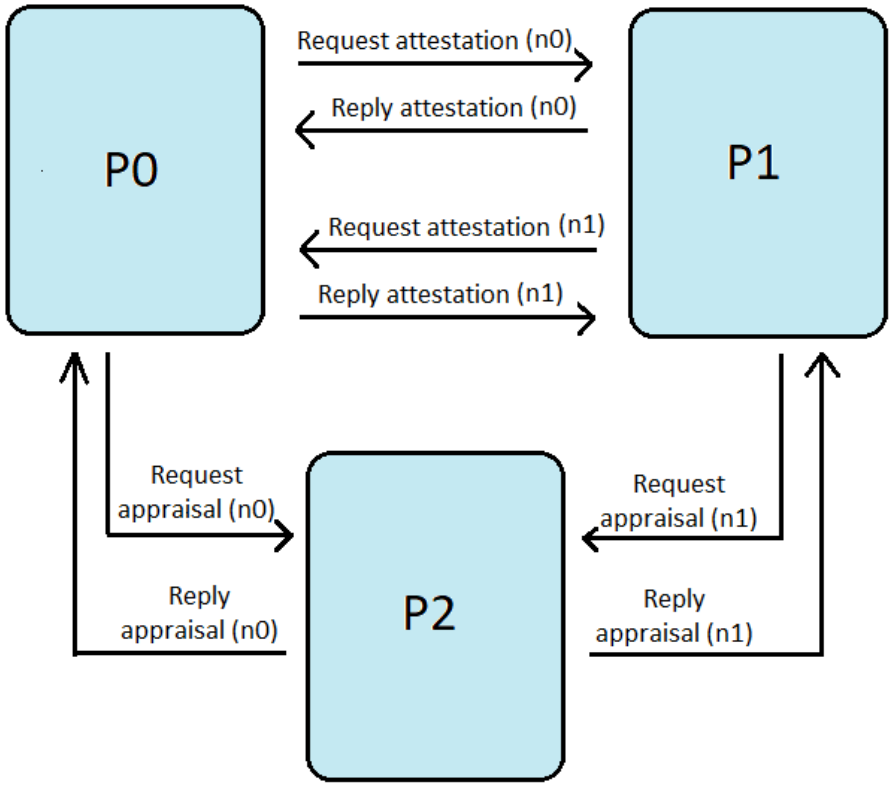
```
* P0, n0: @P1 [ attest01 P1 sys ] ⇒ @P2 [ appraise01 P2 sys ]
* P1, n1: @P0 [ attest10 P0 sys ] ⇒ @P2 [ appraise10 P2 sys ]
```

In the first Copland phrase, P0 starts the attestation process and sends a nonce n0 as initial evidence to guarantee freshness. It subsequently requests P1 to attest its system. The returned evidence is sent to P2 to appraise it. This result is then sent back to P1, which now can make the appropriate trust decisions based on the evidence generated in the process. In the second Copland phrase, the roles of P0 and P1 are inverted.

4.2 Problems with mutual attestation

Although the example shown in section 4.1 can be expressed with the previously shown phrases, it makes use of an odd addition to make the Copland syntax flexible enough. In Copland, an atomic phrase of the form m q t refers to a measurement [20], where m names the measurement itself, q refers to the location (or principal) in which it takes place and t designates its target. A measurement can be defined as an operation that produces evidence based on the state of a target component that can be appraised by a remote party. As an example, in the case of the phrase "attest01 P1 sys", the protocol is executing a measurement named attest01 on the component system located in P1. The appraisal process included in the previous example does not exactly fit into this definition of a measurement and requires some

Figure 4.1: Parallel Mutual Attestation



more context for its role in the protocol to be understood. If we take for instance the phrase "appraise01 P2 sys", following the Copland syntax of a measurement we would say that we are executing a measurement appraise01 on the component system located in principal P2. However, in this case we are not running a measurement on any component of P2, we are running an appraisal of the evidence generated so far by the attestation. Another way of writing the same phrase would be "appraise01 P2 it", which better conveys the idea that the operation is being performed on the evidence received, but also highlights the fact that an appraisal does not follow the syntax for atomic measurements.

In a typical Copland phrase, the appraisal action is left implicit, as we assume that the principal that starts the attestation process will appraise the evidence generated by the measurements in some way before taking a trust decision. But in

the mutual attestation example discussed so far, the appraisal has to be included explicitly because it is performed by a different principal. Furthermore, it does not produce new evidence and only transforms the received evidence into a Boolean or an enumerated set of values that defines if the evidence passed or failed the assessment. In this regard, an appraisal action is more akin to an evidence operator like copy, hash or sign, which are defined terms in the Copland syntax. Considering that Ramsdell et al. [16] affirmed that they expect to add new terms to Copland, a term for an appraisal action would certainly help make some phrases less ambiguous and the Copland syntax more elegant. However, for a protocol like the parallel mutual attestation example described in this section, the anti-pattern of using an appraisal as a measurement works well.

4.3 Embedded mutual attestation

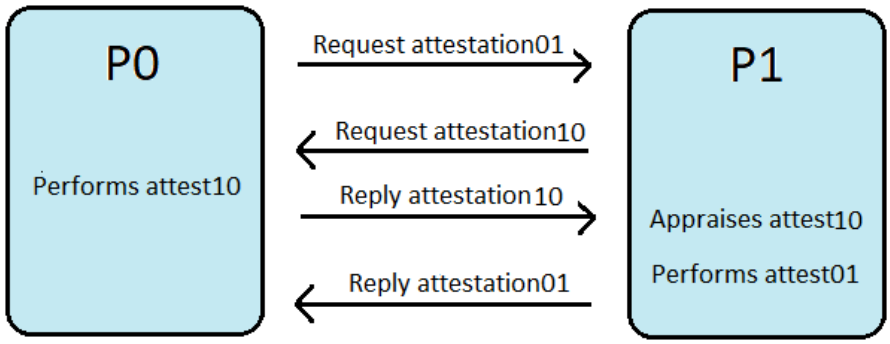
Although there are many mutual attestation protocols that can be expressed as parallel attestations, like the one in the previous example, some are required to be part of a single string of ordered events. We can imagine a server running in a cloud computing service to which a remote station has to connect in order to process some confidential information. To ensure that neither the station nor the server have been compromised by an attacker, a mutual attestation protocol can be implemented. Principal P0 would be the remote station and P1 would be the server. After receiving an initial attestation request by P0, P1 makes its own attestation request to P0 and waits for the evidence to appraise. If the evidence passes the assessment, P1 makes a measurement of its system and sends that evidence back to P0. This ends the attestation, as the final evidence is assumed to be checked by P0 to take the adequate actions afterwards.

We can express this protocol with the following Copland phrase:

```
* P0: @P1 [@P0 [ attest10 P0 sys ] ⇒ attest01 P1 sys]
```

In the phrase, the attestation is started by the remote station P0, which sends a request to the cloud server P0. Before replying to the request, P0 makes its own attestation request to P1, which fulfills it by executing measurement attest10 on the system located in itself. The generated evidence is sent back to P1, which then attests its own system state with the measurement "attest01 P1 sys". This evidence is sent back to P0 and the protocol is finished. The attest measurements in this phrase could actually represent many ordered measurements on each principal or could be even part of a deeper layered attestation. We express them as a single measurement for simplicity. The flow and order of attestations in this phrase can be seen in Figure 4.2.

Figure 4.2: Embedded mutual attestation



In the discussed Copland phrase, the intermediate appraisal made by P1 before attesting its state to P0 is left implicit. According to the Copland Use Cases note written by Kretz et al. and included in the Github repository [10], this is one of the two approaches that can be taken to express these kinds of protocols. However, it is apparent that we are losing valuable information about the protocol by leaving

out of the Copland phrase such an integral portion of it. If we were to explicitly include the embedded appraisal of the attestation, we could write the following, more complex, Copland phrase:

```
* P0: @P1 [@P0 [ attest10 P0 sys ] ⇒ ( appraise P1 it ⇒
{} +<- attest01 P1 sys ) ]
```

In this case, we have adopted the anti-pattern described in section 4.2 of treating an appraisal as a measurement. The evidence produced at the first attestation at P0 is sent back to P1 and then is distributed via the sequential branching operator "$<$ " such that it is consumed by the appraise action to the left and not sent to the second attest measurement to the right. After P0 appraises the evidence, it decides whether or not to continue the attestation protocol. In case it does, it first nullifies the result of the appraise action (as otherwise it would be bundled together with the evidence of the second attest, which is not necessary) and then executes the attestation of its own system. The generated evidence is sent back to P0 and the protocol concludes.

Although we already discussed that this use of a measurement is not what this kind of phrase was meant to express, there is a bigger difference in this case. In our protocol, the embedded attestation requires P1 to make a trust decision in the middle of the protocol that can stop its execution based on the result of the appraisal. This conditional is a behavior that cannot be expressed with Copland. In a regular Copland phrase, a series of measurements are performed in some unalterable succession (although depending on the operators, some of them might run in parallel) and a decision point only comes after the protocol has concluded and the principal that started the attestation appraises the generated evidence, an assessment that is not part of the attestation itself. In this case, however, we have a different kind of

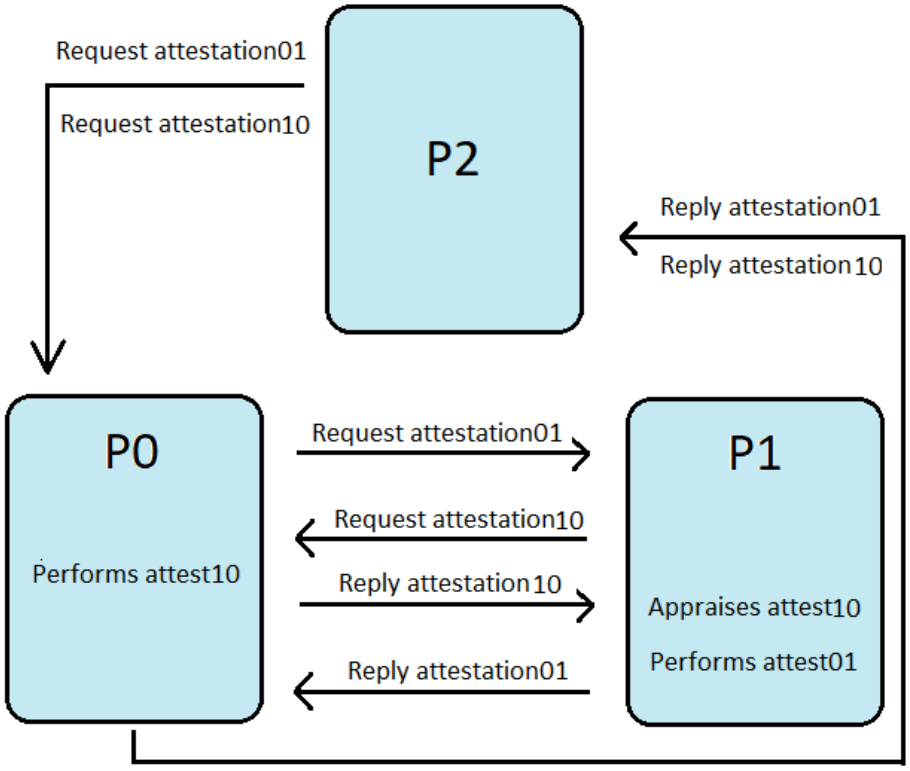
action with an atypical behavior that cannot be conveyed by the Copland phrase itself, but by the context that we attach to the protocol. This could lead to different results in a static analysis that identifies attack scenarios in which an adversary successfully avoids detection by the protocol.

We can make a small modification to the previously discussed embedded mutual attestation example to illustrate this problem. Let us imagine a situation in which a central in-house server P2 is the one that bootstraps the attestation process and appraises the final evidence on P1 gathered by P0, as shown in Figure 4.3. Before the protocol concludes, P2 has to measure its state, again, this time for P2 to appraise. We can express this modified protocol by adding some minimal changes to our phrase:

```
* P2: @P0 [ @P1 [(@P0 [attest10 P0 sys] ⇒ (appraise P1 it ⇒
{} +<- attest01 P1 sys))] -<- attest10 P0 sys]
```

In this example, an attacker avoids detection at P0 's system if the system is corrupted by the time the final attest10 measurement is performed. To prevent replay attacks by P0, a nonce should be included at the beginning of the attestation. As we mentioned before, this attestation measurement can be seen as an abstraction of a deeper layered attestation process. Without external context on the protocol, an analysis would disregard the appraisal done by P1, as the evidence generated by it is nullified. Thus, an attack scenario in which P0 is corrupted before the protocol begins would be valid in theory, as the attacker could then corrupt the attest10 process just before the last measurement and avoid detection. However, a corrupted P0 would not pass the appraisal done by P1. In the security analysis, this would be seen as an irrelevant fact, but as P0 has the power to stop the attestation process altogether, it would not send its own attestation evidence back to P0, which would

Figure 4.3: Embedded mutual attestation modified



prevent the attacker from avoiding detection at P0 's system as P2 would notice that something is wrong if it only receives P0 's evidence.

As the previous example showed, the lack of a way to express certain nuances in mutual attestation scenarios on Copland makes the inclusion of context an imperative to fully grasp a protocol. However, one of the purposes of Copland is to be able to express these kinds of scenarios with its syntax. The problem is also highlighted by the use of the Chase model-finder tool, as there is no way in the Copland phrase to express the decision point in the middle of the protocol. To model SGX's attestation successfully, we include additional assumptions to our Chase inputs, which will be discussed in chapter 5.

Chapter 5

Analysis of Attestation in SGX

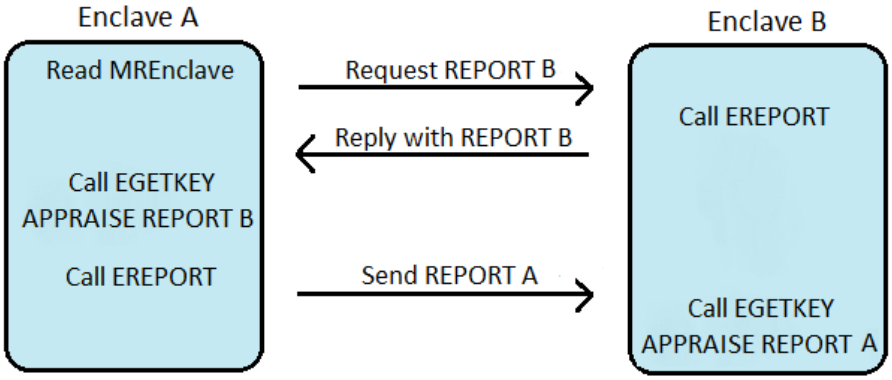
In this chapter, we present our analysis of the three SGX attestation protocols: Local attestation, Platform Provisioning and Remote attestation. For each of them, we present our modeled phrase in Copland, the attack approaches found with the Chase model-finder and the security assumptions that the protocols requires to be considered safe.

5.1 Local attestation

Local attestation is a process in Intel's SGX architecture in which two enclaves located in the same system verify each other's identity and confirm that they are indeed running in the same machine. The general actions performed during the process can be seen in Figure 5.1.

The analysis approach we take for SGX was first presented by Rowe et. al for generic attestation protocols [20]. The first step in our analysis is to describe SGX's local attestation process and its components with a Copland phrase. The main principals that interact in the protocol are the two enclaves that will attest to each other, which are named encA and encB in our phrase. Figure 5.2 shows a basic

Figure 5.1: Information flow in SGX's local attestation. The enclaves are located on the same system.



Copland phrase that expresses the protocol.

5.1.1 Copland Phrase Details

The attestation is initiated by encA, which takes the role of the appraising party. It then retrieves its mrenclave identity record. This section is expressed by: $*$encA : read encA mrenclaveA. The retrieval of enclave A's mrenclave is performed by the measurement read, which takes place on encA. The retrieved mrenclave is sent to encB via the $\rightarrow$ operator, which stands for sequential execution, with a request to attest its state.

```
*encA: read encA mrenclaveA
    ⇒ @encB [ereport encB stateB] ⇒ (egetkey encA stateA -<+ _)
    ⇒ appraise encA reportB ⇒ {} ⇒ ereport encA stateA
    ⇒ @encB [(egetkey encA stateA -<+ _)
        ⇒ appraise encB reportA ]
```

Figure 5.2: First Copland phrase for SGX's local attestation

Once encB receives the request from encA with @encB[...], it calls the EREPORT CPU leaf instruction, used to obtain a REPORT of encB 's identity binded with encA 's mrenclave. This REPORT is then sent back in a reply to encA, which runs

two operations: the first one is the CPU leaf instruction EGETKEY, called with the measurement msp(egetkey, encA, stateA) and which allows the enclave to obtain the report key needed to validate the REPORT sent by encB. The second operation is a copy of this REPORT, which is expressed in Copland with the operator _ and that receives the evidence thanks to the sequential branching done with $-<+$. We branch our operations because we want both pieces of evidence (the REPORT and the report key) to be bundled together before being passed on to the next operation, which is the verification of encB 's state done by encA. In our phrase, we express it with the measurement msp(appraise, encA, reportB), and then we null the produced evidence with the shorthand {}, as its result is used by encA to decide whether or not encB can be trusted but has no use for the rest of the protocol. Notice that we have to make the evidence appraisal at encA explicit by adapting it in a Copland atomic measurement action in order to express the trust decision at the middle of the protocol, as it can be stopped if encA detects a corruption in encB.

After encA has verified the identity of encB, it also calls the EREPORT instruction to produce a REPORT of its own state so that encB can appraise encA 's identity as well. This is expressed with the same Copland measurements used in the previous step, but performed on the opposite enclave. We include the final appraisal explicitly because otherwise the final evidence bundle would be returned to encA, as it is the principal that started the protocol. But as local attestation in SGX is an example of mutual attestation in which each appraisal is done in succession, we want the second enclave to verify the identity of the first one before the protocol returns to encA and is finalized.

Although this phrase correctly describes the protocol, we can make a change in order to better convey its nuances. Both EGETKEY and EREPORT are called from within the enclaves, as it is a requirement imposed by Intel. But as they are CPU

instructions, besides the inputs, their execution cannot be influenced in any way from the enclaves. Thus, we consider it helpful to make this distinction explicit in this case by introducing a third principal, that we call cpu and that is the one that performs the egetkey and ereport measurements after receiving a request from one of the enclaves. We believe this can make it easier to digest before we move on to more complex attestations in SGX in which we leave this relation implicit. This modified Copland phrase can be seen in Figure 5.3.

```
*encA: read encA mrenclaveA ⇒ @encB [ @cpu[ereport encB stateB] ]
    ⇒ (@cpu[egetkey encA stateA] -<+ _)
        ⇒ appraise encA reportB ⇒ {}
    ⇒ @cpu[ereport encA stateA]
    ⇒ @encB [(@cpu[egetkey encA stateA] -<+ _)
        ⇒ appraise encB reportA ]
```

Figure 5.3: Modified Copland phrase for SGX's local attestation

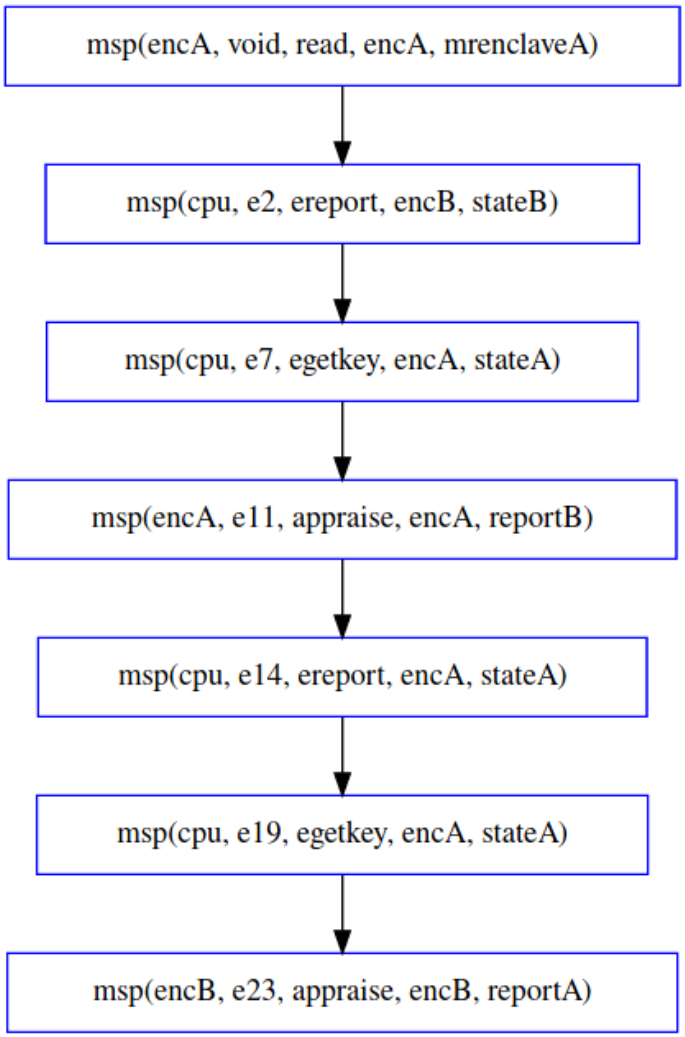
The Copland phrase can be translated into an event system that determines the ordering of the measurements in the protocol. Figure 5.4 shows the event system for our SGX's local attestation phrase, also generated by Chase. The system shows that every operation during the local attestation process is executed in a strict linear order. Considering the fact that it would be easy to modify the protocol so that the attestations could run in parallel (as they do not depend on the results of the other), the decision to have them run sequentially seems to be a design choice by Intel. The label of each event follows the pattern "e0", where 0 increases for each event according to the partial ordering established by the Copland phrase. Besides the measurement events, additional events take place when a principal requests measurements from other principals, with the request and the subsequent reply acting as separate events. The branching operators also create two additional events, one for splitting the input evidence and one for joining the resulting evidence. The

events in SGX's local attestation are the following:

- l(e0) = msp(encA, void, read, encA, mrenclaveA)
- l(e1) = req(encA, encB)
- l(e2) = req(encB, cpu)
- l(e3) = msp(cpu, e2, ereport, encB, stateB)
- l(e4) = rpy(encB, cpu)
- l(e5) = rpy(encA, encB)
- l(e6) = split(encA, -, <, +)
- l(e7) = req(encA, cpu)
- l(e8) = msp(cpu, e7, egetkey, encA, stateA)
- l(e9) = rpy(encA, cpu)
- l(e10) = cpy(encA)
- l(e11) = join(encA)
- l(e12) = msp(encA, e11, appraise, encA, reportB)
- l(e13) = nul(encA)
- l(e14) = req(encA, cpu)
- l(e15) = msp(cpu, e14, ereport, encA, stateA)
- l(e16) = rpy(encA, cpu)
- l(e17) = req(encA, encB)
- l(e18) = split(encB, -, <, +)
- l(e19) = req(encB, cpu)

- $\mathsf{l}(\mathsf{e20}) = \mathrm{msp}(\mathsf{cpu}, \mathsf{e19}, \mathsf{egetkey}, \mathsf{encA}, \mathsf{stateA})$
- $\mathsf{l}(\mathsf{e21}) = \mathrm{rpy}(\mathsf{encB}, \mathsf{cpu})$
- $\mathsf{l}(\mathsf{e22}) = \mathrm{cpy}(\mathsf{encB})$
- $\mathsf{l}(\mathsf{e23}) = \mathsf{join}(\mathsf{encB})$
- $\mathsf{l}(\mathsf{e24}) = \mathrm{msp}(\mathsf{encB}, \mathsf{e23}, \mathsf{appraise}, \mathsf{encB}, \mathsf{reportA})$
- $\mathsf{l}(\mathsf{e25}) = \mathrm{rpy}(\mathsf{encA}, \mathsf{encB})$

Figure 5.4: Event system for SGX's local attestation.



With the event system obtained we can start our attack analysis with Chase [20]. As we are working with a mutual attestation scenario, an attacker could potentially corrupt any of the two enclaves and each of these cases entails its own analysis, although in practice they will be very similar because the attestations work almost like a sequential mirror of each other. To analyze the attack scenarios to one of the enclaves we took a portion of the original Copland phrase, which can be seen in Figure 5.5 and corresponds to the first part of the attestation, when encB has finished attesting its state to encA but encA still has not attested itself to encB.

```
*encA: read encA mrenclaveA
    ⇒ @encB [ @cpu[ereport encB stateB] ]
    ⇒ (@cpu[egetkey encA stateA] -<+ _)
    ⇒ appraise encA reportB
```

Figure 5.5: Analyzed portion of SGX's local attestation

The first step in the analysis is determining the measurement of interest, which is the point at which we want to find whether an attacker has compromised a certain component of the system without being detected. We define our measurement of interest in Figure 5.6. The first part of the logical statement defines e11 as the measurement event at which the attacker would be considered successful if it had compromised the system without being detected. The second part of the statement designates the corruption target of the attacker and is defined with the phi predicate. In our phrase, the target is reportB, which includes information about the identity of encB to be appraised and whose corruption could imply that the enclave has been compromised or that the attestation request was fulfilled by a simulated enclave.

The Chase model finder produces a series of possible attack models based on the Copland phrase and the parameters provided at runtime. One more assumption that we include for the initial analysis is the dependencies of the appraisal action,

```
l(E) = msp(encA, e11, appraise, encA, reportB) ⇒
phi(encA, reportB, E)
```

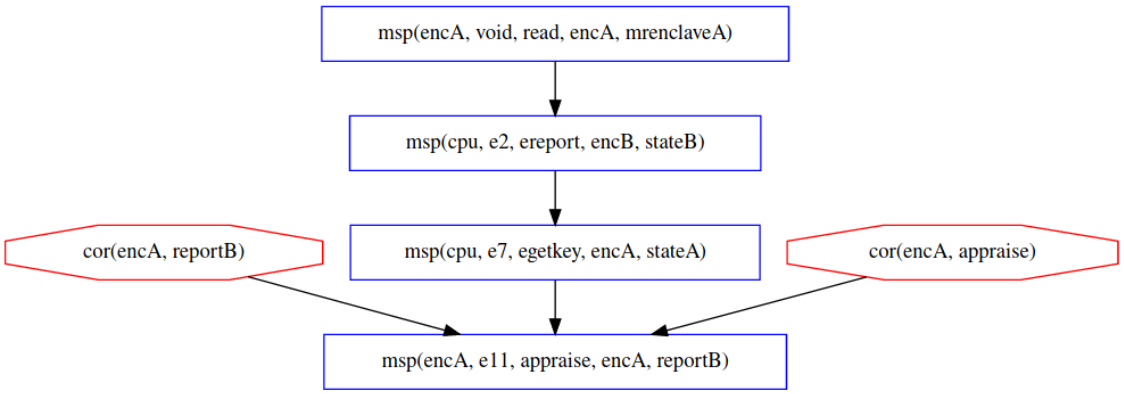
Figure 5.6: Measurement of interest for the reworked SGX phrase

```
depends(P, C, encA, appraise) ⇒ P = cpu ∧ C = egetkey ∨
P = cpu ∧ C = ereport ∨ P = encB ∧ C = stateB ∨ P = encA ∧
C = c.
```

Figure 5.7: Dependencies of the appraisal process

shown in Figure 5.7. We define a relation of dependency between the appraise measurement and four different components: the egetkey measurement, located in cpu principal; the ereport measurement, also in the cpu; the stored state of encB; and a "c" component located in encA that stands for any unaccounted dependency not explicitly included in the Copland phrase but that might affect the appraisal. By running Chase with the selected phrase and measurement of interest, we obtain a total of 11 attack scenarios. One of them can be seen in Figure 5.8. In this case, an attacker corrupted the identity of encB and then corrupted the appraisal method employed by encA to verify the encB 's REPORT.

Figure 5.8: Example of an attack scenario in which an attacker would beat the attestation.



5.1.2 Results

By analyzing the attack scenarios produced by the Chase model finder we can find three general categories of attacks. These results are shown in table 5.1.

Attack category	Incidence
Scenarios in which the EGETKEY or the EREPORT leaf instructions have been attacked	10
Scenarios in which the appraisal process has been attacked	1
Scenarios in which an unknown dependency in the attestation has been attacked	1

Table 5.1: Attack categories for partial Local attestation

Hardware considerations can further reduce the amount of attack scenarios based on the system's architecture. Figure 5.9 states that no component that runs directly on the cpu can be corrupted. This is a reasonable assumption when we consider that an attack of the EGETKEY or EREPORT implementations (to make them, for instance, be allowed to be called from outside an enclave) would require hardware modifications that would be impractical for an attacker. By applying this assumption, the number of attack scenarios is reduced to 2.

```
l(E) = cor(cpu, C) ⇒ false
```

Figure 5.9: Assumption to forbid the corruption of components at cpu

Analyzing the entire local attestation phrase follows a similar process. Our measurement of interest focuses on the appraise action on reportA, identified as event e27. We look at any scenario in which a corrupted reportA, which stores the appraised enclave's identity, fools detection by the time the appraise action takes place. As this time we are considering the entire phrase, we need to include an additional dependency for the conditional in the first part of the protocol. We add

the appraise action from encA as a direct dependency of encB and include its relations to other components in both principals. Figure 5.10 shows the measurement of interest and the assumptions for this protocol.

```
l(E) = msp(encB, e26, appraise, encB, reportA) ⇒
phi(encB, reportA, E).

% Assumptions about system dependencies.
depends(P, C, encB, appraise) ⇒ P = encA ∧ C = appraise ∨
P = cpu ∧ C = egetkey ∨ P = cpu ∧ C = ereport ∨ P = encB ∧
C = stateB ∨ P = encB ∧ C = c.
depends(P, C, encA, appraise) ⇒ P = cpu ∧ C = egetkey ∨
P = cpu ∧ C = ereport ∨ P = encB ∧ C = stateB ∨ P = encA ∧
C = c.
```

Figure 5.10: Assumptions for complete local attestation phrase

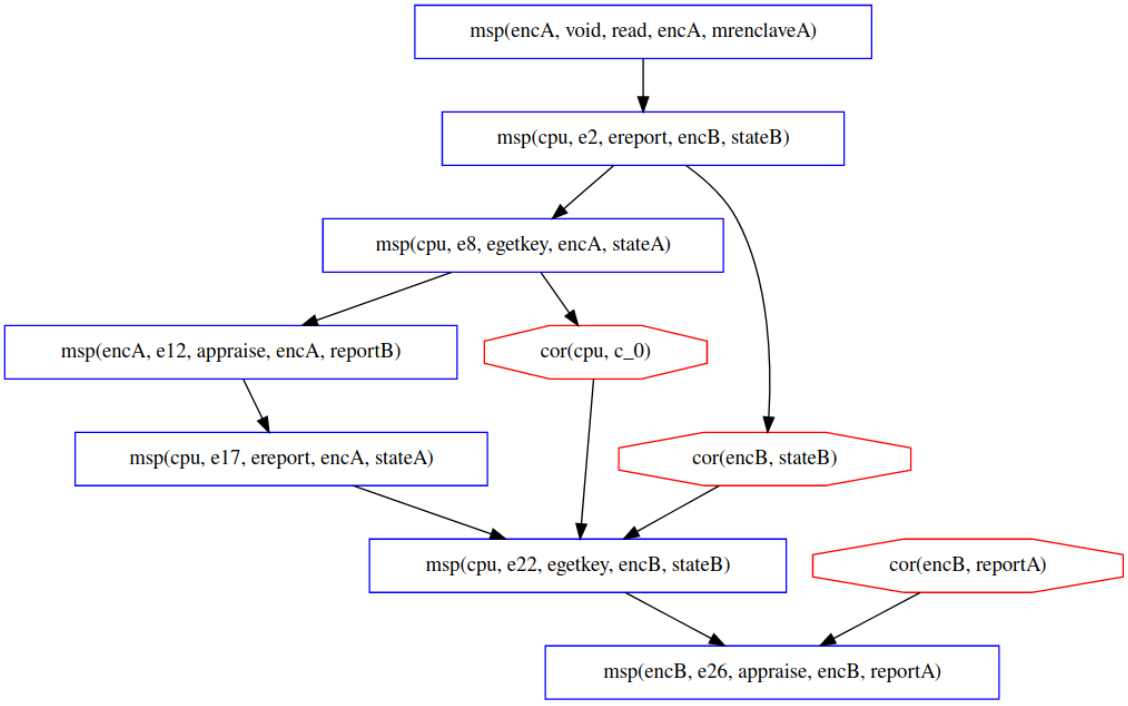
By executing Chase with the provided input we obtain 67 different attack models. After analyzing the phrases, we found the the corruption scenarios fall into four scenarios, shown in table 5.2.

Attack category	Incidence
Scenarios in which the EGETKEY or the EREPORT leaf instructions have been attacked (could include instances of the other categories)	55
Scenarios in which the appraisal process has been attacked	1
Scenarios in which an unknown dependency in the cpu has been attacked	3
Scenarios in which the state of the second enclave has been attacked	8

Table 5.2: Attack categories for Local attestation

An example of the third category is shown in Figure 5.11, in which an attacker successfully corrupts the identity of the first enclave by attacking part of the context of the EGETKEY leaf instruction on the cpu and then the state of the second enclave in order to be able to pass the appraise action with a corrupted target enclave. In practice, this kind of attack would be unfeasible, as the attacker would need to

Figure 5.11: Attack scenario for the complete local attestation.



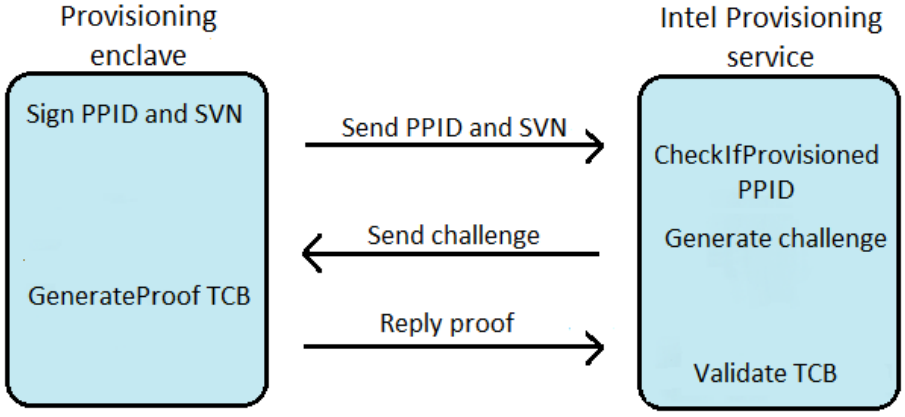
corrupt an unidentified component in the cpu principal that would affect the key retrieval instruction in such a way that the corrupted enclave would pass appraisal. As before, we can limit these kinds of attacks by including the assumption shown in figure 5.9 and reduce our attack scenarios to three instances.

5.2 Platform Provisioning

In order for an enclave to prove their identity to an external Service Provider during remote attestation with SGX, the target enclave has to send a verifiable cryptographic QUOTE. However, a QUOTE can only be produced by an enclave once it gains access to a special asymmetric attestation key. The process by which a SGX-enabled machine obtains this key is known as Platform Provisioning. It only takes place once and the obtained key is stored and used in all subsequent remote

attestations. The only exception takes place whenever a new security update becomes available, in which case the Platform Provisioning protocol is executed again. Figure 5.12 gives a glimpse on the Platform Provisioning protocol.

Figure 5.12: Information flow during Platform Provisioning.




```
*ips, ipspubk:
    @pve [((read pve pk1 ⇒ # ⇒ !) +~+ (read pve svn ⇒ !))]
    ⇒ checkIfProvisioned ips ppid ⇒ genChallenge ips pk2
    ⇒ @pve [(decrypt pve chal ⇒ genProof pve tcb)]
    ⇒ validate ips tcb2
```

Figure 5.13: Copland phrase for Platorm Provisioning

5.2.1 Copland Phrase Details

Figure 5.13 presents our Copland phrase for Platform Provisioning. Two principals interact during the protocol: Intel's Provisioning Service (named ips in our phrase) and the provisioning enclave (named pve in our phrase), The later is a special enclave that comes preinstalled in every SGX-capable machine and only interacts with with this protocol. At the beginning of the process, the pve uses the IPS's public key (ipspubk) to sign the hash of the Platform Provisioning ID (pk1) and its Security

Version Number (svn). Both are then returned to the IPS, which checks if a provisioning has already taken place for a particular Platform Provisioning ID (ppid). Next, the IPS creates a challenge based on the information received by the PVE and sends it to the pve. The pve decrypts the challenge and generates a proof based on its Trusted Computing Base (tcb) and then returns it to the IPS for validation, which proves the identity of the machine to the IPS.

Of the two entities that participate in this protocol, our analysis will only consider the case in which the pve is compromised or fake, as we can assume the IPS will be secure thanks to its location inside Intel's premises and also because its security is not guaranteed by the SGX implementation, which the focus of this work.

The Platform Provisioning protocol illustrates another possible improvement to the Copland language: a way to name the evidence we get from a measurement, so that in case that evidence is going to be used in the next measurement, the language can express this relationship explicitly. Consider the last two measurements in our Copland phrase. The first one generates proof that satisfies the IPS's challenge based on its Trusting Computing Base (tcb). The evidence generated returns to the IPS, which decrypts it and verifies its correctness. In the Copland phrase, we name this component tcb2. However, Copland lacks a mechanism that allows us to specify that this tcb2 comes directly from the evidence produced in the previous measurement, so in order to express this relationship in the Chase model finder, we have to add a dependency assumption, which can be seen in Figure 5.10.

```
depends(P, C, ips, tcb2) ⇒ P = pve ∧ C = genProof ∨ P = pve ∧
C = tcb.
```

Figure 5.14: Basic dependencies for Platform Provisioning

With this assumption, we specify that tcb2 explicitly depends on the tcb com-

ponent measure previously and the genProof process itself. However, if there was a way to express that the previous evidence is the target of the next measurement in the Copland phrase itself, the protocol would avoid ambiguity and we would not need to add these kinds of assumptions.

In order to express the rest of the relationships in the protocol, we add the following five dependencies:

- depends(P, C, ips, tcb2) → P = pve & C = genProof ‖ P = pve & C = tcb.
- depends(P, C, pve, tcb) → P = ips & C = ppid.
- depends(P, C, ips, ppid) → P = pve & C = pk1 ‖ P = pve & C = svn.
- depends(P, C, pve, genProof) → P = ips & C = genChallenge.
- depends(P, C, pve, genChallenge) → P = ips & C = checkIfProvisioned ‖ P = ips & C = ppid.

The first dependency indicates that the genProof process depends on the challenge sent by the IPS. The second one signals that the challenge received by the pve depends on the challenge generation process at the IPS. The last one indicates that the challenge regeneration depends on the reading process and on the components platforming keys and svn.

For our last step, we define the measurement of interest in our analysis, which in this case is the validate measurement at Intel's Provisioning Service. The target component is the tcb. This definition can be seen in Figure 5.15.

5.2.2 Results

By running Chase on our Copland phrase with the stated assumptions, we obtain 110 different attack scenarios for the Platform Provisioning process. An example of

```
l(E) = msp(isp, E1, validate, isp, tcb2) ⇒ prec(E, E2) ∧
phi(pve, tcb, E2).
```

Figure 5.15: Measure of interest for Platform Provisioning

an attack model can found in Figure 5.17. In this attack, the adversary is able to fool detection by corrupting, besides the trusted computing base, the proof generation process itself that takes place in the provisioning enclave,

Furthermore, we can classify the attack scenarios into three categories. They are included in Table 5.3.

Attack category	Incidence
Attacks that take place after the protocol has finished	32
Attacks that corrupt some of the processes or inputs in the provisioning enclave that produce the response to the IPS	9
Attacks that corrupt some of the validating process in the IPS	69

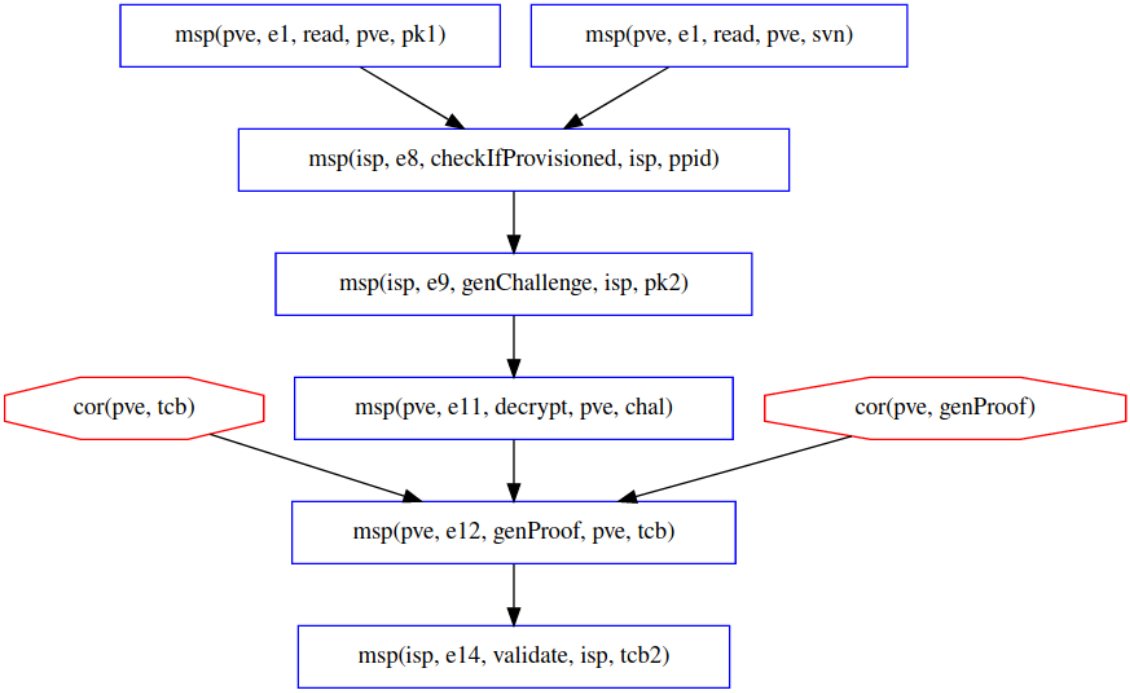
Table 5.3: Attack categories for Platform Provisioning

Although the number of possible attack scenarios is considerable, we can reduce it by applying a number of assumptions based on this particular case. For instance, the attack scenarios that take place after the protocol's last measurement are not of concern in this case, as they would still need to pass SGX remote attestation before the platform receives any kind of sensitive information that an attacker might be seeking. We can exclude all scenarios in which a corruption event happens after our final measurement event on the quoting enclave, whose label is e13 in our Copland phrase, by applying the following assumption:

```
prec(e13, E2) ∧ l(E2) = cor(P,C) ∧ ms_evt(e13) ⇒ false.
```

Figure 5.16: Assumption for Platform Provisioning

Figure 5.17: Attack scenario for Platform Provisioning.



This reduces the number of attack scenarios to 78. The next assumption is related to the security of the IPS. As attacks on large tech companies over the past decade have demonstrated, no organization can guarantee that their security level would be enough to prevent possible cyberattacks. A successful attack on the IPS could bring more serious problems than the integrity of the provisioning process itself, as it would compromise every SGX-compatible chip's root provisioning keys, which are created during the manufacturing process and stored by Intel to confirm the chip's identity. SGX technology, however, also makes use of the root sealing key to derive keys used during remote attestation. The root sealing key is generated the first time the device is turned on and thus is not saved by Intel. This reduces the security risks of a successful attack on the IPS, although the provisioning process itself would remain compromised. This is a clear weakness of the SGX design, even with the provisions added by Intel to mitigate its impact.

After considering the risks involved with any kind of attack on the IPS, we can exclude them from our results by defining an assumption in which components located inside the IPS cannot be corrupted. We can accomplish this with the following statement:

`l(E) = cor(ips, C)` $\Rightarrow$ `false.`

By applying this assumption the number of attack scenarios is reduced to nine. At this point, we can conclude that the weakest link of the Platform Provisioning process is the Provisioning enclave. The security of this enclave is high, as it cannot be accessed by users and it is only involved with in this process. By including an assumption to forbid attacks against the pve, the attack scenarios get reduced to zero. However, additional research is required to confirm is this enclave is as secure as required to prevent attacks, as a vulnerability in it would leak the attestation key used by the platform in SGX's remote attestation.

5.3 Remote attestation

After an SGX-enabled device has obtained its attestation key through the Platform Provisioning process described in section 5.2, it can attest itself to external entities through remote attestation. In order to achieve it, a special enclave known as the Quoting enclave must create a QUOTE of the target enclave signed with the device's attestation key so that it can be verified by Intel. The Quoting enclave comes preinstalled by default in every SGX-enabled device and its sole purpose is to facilitate remote attestation, as is the only enclave that can access the attestation key and thus helps protect it even from other enclaves [5]. Figure 5.18 presents our Copland phrase for remote attestation.

```
*sv: @isp [read1 isp epidGroup] ⇒ @ias [read2 ias sigRL]
  ⇒ genChallenge sv spid
  ⇒ @isp [ereport isp stateISV
           ⇒ @qe (egetkey qe stateQE -<+ _)
                 ⇒ verify isp stateISV ⇒ egetkey qe epidkey
                 ⇒ genQuote isp mrenclave]
  ⇒ @ias [(validate ias quote +<+ checkprivrl ias quote)
          ⇒ createReport ias report]
```

Figure 5.18: Copland phrase for Remote attestation

5.3.1 Copland Phrase Details

The attestation is initiated by the Service Vendor (sv in our phrase), which sends a request to the Independent Service Provider's (isp) enclave for its alleged EPID group in the measurement @isp [read1 isp epidGroup]. In this scenario, the sv asks the isp to prove its identity before it can share company secrets with it. After the sv receives the EPID group, it asks the Intel Attestation Service ias for an updated Signature Revocation List sigRL with the next part of the Copland phrase: @ias [...].

After receiving a reply from Intel, the isp generates a challenge message with the measurement genChallenge that includes the updated sigRL, its Service Provider ID spid and a nonce for freshness. Once the enclave receives the challenge, it calls the EREPORT leaf instruction to generate a cryptographic REPORT and sends it to the Quoting enclave qe via the => linear sequence operator. At this point, a local attestation process starts in which the Independent Service Provider's enclave attests itself to the Quoting enclave. The qe calls the measurement msp(egetkey qe stateQE) to obtain the report key necessary to decrypt the target's enclave REPORT. It then verifies the isp's identity.

Once the Quoting enclave finishes local attestation, it calls the EGETKEY instruction again, but this time to retrieve the attestation key (also known as EPID

private key) obtained during Platform Provisioning. We express this operation with the measurement msp(egetkey qe epidkey). With the key in its possession, the Quoting enclave can finally generate the QUOTE over the isp's MRENCLAVE identity signature and encrypt it with the Intel Attestation Service's public key, which is hard-coded in the qe. We abstract these operations with the msp(genQuote qe mrenclave) measurement.

The generated QUOTE is sent back to the isp, then to the Service Vendor and finally to Intel Attestation Service. The ias performs two verification: it first validates the QUOTE based on its identity signature and then checks that the device is not included in the Private Revocation List. If it was able to confirm the platform's identity, it creates an attestation report with the measurement msp(createReport ias report) and sends it to the Service Vendor, which finishes the protocol.

```
l(E) = msp(qe, E1, genQuote, isv, mrenclave) ⇒
phi(isv, mrenclave, E).
```

Figure 5.19: Measurement of interest for Remote attestation

For our analysis, we consider the case in which an attacker tries to compromise the Independent Service Provider's enclave, as it is the principal that would receive the information the attacker would potentially want to get access to. Our measurement of interest cam be seen in Figure 5.19 and is the genQuote operation on the Quoting enclave and the component the attacker would need to corrupt to be considered successful is the identity signature MRENCLAVE, which describes the state of the target enclave.

After modeling the relations between the different components of the protocol, we came up with the following dependencies:

- depends(P, C, ias, report) → P = ias & C = validate ‖ P = ias & C = checkprivrl ‖ P = ias & C = quote.
- depends(P, C, ias, quote) → P = qe & C = genQuote ‖ P = isp & C = mrenclave.
- depends(P, C, ias, quote) → P = isp & C = mrenclave ‖ P = isp & C = stateISV.
- depends(P, C, isp, mrenclave) → P = isp & C = stateISV.
- depends(P, C, qe, genQuote) → P = qe & C = egetkey ‖ P = isp & C = stateISV.
- depends(P, C, qe, verify) → P = qe & C = egetkey ‖ P = isp & C = ereport.
- depends(P, C, qe, egetkey) → P = qe & C = stateQE ‖ P = qe & C = verify.

The first three dependencies refer to the evidence flow to the components of the Intel Attestation Service. But as our measurement of interest takes place before the QUOTE is sent to the ias, its components do not participate in any attack analysis. The fourth dependency sets an explicit relation between the enclave's identity signature and the state of the enclave itself. The quote generation process depends directly on the result of the EGETKEY instruction to get the attestation key and of the state of the target enclave. The next dependency refers to the Quoting enclave verification during local attestation, and it depends on the result of the EREPORT instruction and of the EGETKEY to get the report key. Our last statement is used to express the conditional step that we cannot express in Copland directly (as explained in section 4.3) and which refers to the result of the embedded local attestation that can stop the protocol in case it cannot be verified.

5.3.2 Results

After running Chase with the detailed inputs, we obtain 114 different attack scenarios against Remote attestation. By analyzing the results, we can identify three attack categories that an adversary could try, which are included in Table 5.4.

Figure 5.20: Attack scenario for Remote attestation.

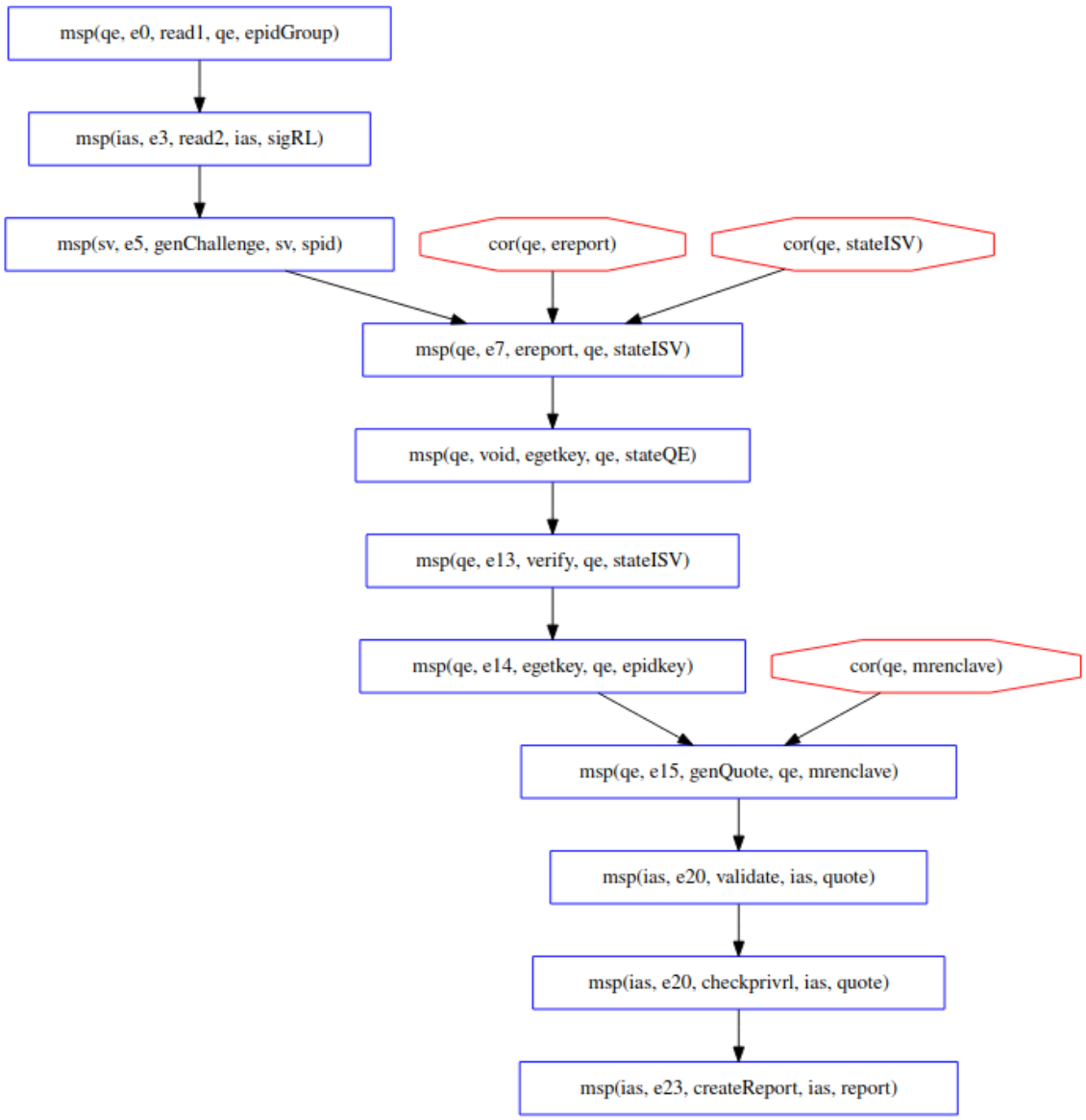


Figure 5.20 shows an example in which an attacker avoids detection by corrupting the EREPORT instruction so that the compromised enclave would appear to

Attack category	Incidence
Attacks where the verification or quote generation processes in the Quoting enclave are compromised	25
Attacks against the EGETKEY or EREPORT leaf instructions	67
Attacks against an unidentified dependency in the Quoting enclave	22

Table 5.4: Attack categories for Remote attestation

be working correctly. As we discussed before, many of the found attacks are not actually realistic, particularly the ones that depend on corrupting the processor. In order to account that, we can include a couple of dependencies to specify that the processor leaf instructions ca not be compromised during any kind of attack. These assumptions can be seen in Figure 5.21.

```
(E) = cor(P, egetkey) ⇒ false.
l(E) = cor(P, ereport) ⇒ false.
```

Figure 5.21: Assumptions to limit processor attacks

By applying these assumptions and re-running our Copland phrase through Chase, we can reduce the number of attack scenarios to 47. This is still a considerable number of attack strategies, although they follow similar patterns. In particular, all the identified attack scenarios require the Quoting enclave to be compromised. As this enclave is the only one that can access the attestation key and the one that generates the QUOTE to be verified by Intel, the security of the protocol is very high. It is apparent that the inviolability of the qe is an essential component of SGX's threat model. This is shown by applying the assumption in Figure 5.22, which reduces the attack scenarios to zero. However, different attacks found in the previous years point out that these assumptions were too optimistic.

```
l(E) = cor(qe, C) ⇒ false.
```

Figure 5.22: Assumption to limit attacks against Quoting enclave

Chapter 6

Conclusions

In this work, we analyzed the security of SGX's design with the help of the Copland declarative language and the Chase model-finder. We now present our conclussions for both SGX and Copland.

6.1 SGX

We used Copland to model the three SGX attestation processes: local attestation, platform provisioning and remote attestation, and found the attack approaches that an adversary could pursue in order to compromise the process. We grouped the approaches in different categories. These results are detailed in Chapter 5.

In the case of local attestation, we found the following approaches:

1. Attacks in which the EGETKEY or the EREPORT leaf instructions have been compromised (could include instances of the other categories).

2. Attacks in which the appraisal process has been compromised.

3. Attacks in which an unknown dependency in the processor has been compromised.

4. Attacks in which the state of the second enclave has been compromised.

During our analysis of Platform Provisioning, we found the following attack categories:

1. Attacks that take place after the protocol has finished.
2. Attacks that corrupt some of the processes or inputs in the provisioning enclave that produce the response to the IPS.
3. Attacks that corrupt some of the validating process in the IPS.

Finally, with remote attestation, we found that an attacker could follow these approaches:

1. Attacks where the verification or quote generation processes in the Quoting enclave are compromised.
2. Attacks against the EGETKEY or EREPORT leaf instructions.
3. Attacks against an unidentified dependency in the Quoting enclave.

Next, we tested out different assumptions in each protocol as input for Chase in order to get a better idea of the requirements over which SGX's security rests upon. For instance, we found the following assumptions to be vital for SGX to work as advertised:

- The processor (including the fused-at-production secret keys on it) cannot be compromised.
- The special Quoting and Provisioning enclaves cannot be compromised.
- Intel has to be trusted.

Additionally, we found some design choices that make SGX harder to attack. For instance, we noticed that SGX's protocols have been created to run explicitly in sequence to reduce the possible attack windows, as some of its operations could work in parallel. Also, the fact that the operations that manipulate the most critical keys in a SGX device are handled by separate entities to which users have no direct access reduces the likelihood of attacks. The fact that most attacks on the remote attestation protocol can only succeed by compromising the Quoting enclave first is an indicator of this design choice.

6.2 Copland

Regarding Copland, we ran into issues while attempting to model the complexities of SGX's attestation protocols. As a result, our Copland phrases were missing relevant information about the protocols that we could only be expressed through some external context in addition to the phrases. We find that, although in most cases there are workarounds to model these processes, an extension to Copland would help achieve the language goal of expressing attestations more naturally. We explain many of these considerations and some approaches to deal with them by going through small examples in Chapter 4. Some of the inconveniences we found in Copland are the following:

1. Conditional constructs in mutual attestations are not available in Copland.
2. There is no way to reference previously generated evidence in the phrases.
3. Explicit appraisals have to be expressed as measurements even though they do not comply with their definition.

In most cases, these inconveniences can be solved with workarounds. We also

showed that failing to address them can lead to unrealistic attack scenarios after the phrases were analyzed with Chase. Proposals to fix these cases include dividing complex phrases into smaller ones that can be analysed separately or adding more input assumptions in order to treat them as dependencies.

We also ran into issues when processing very complex Copland phrases with the Chase model-finder, specifically out-of-memory errors. To circumvent this, we reduced the complexity of the system by leaving some operations and principals implicit, like the processor in the case of the leaf instructions. We found this to be a good approach, although it requires including additional assumptions to Chase to describe the operations and their security correctly.

Bibliography

[1] Anati, I., Gueron, S., Johnson, S., and Scarlata, V. Innovative Technology for CPU Based Attestation and Sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy* (2013, June), vol. 13, ACM, p. 7.

[2] Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., and Zaharia, M. A View of Cloud computing. *Communications of the ACM 53*, 4 (2010), 50–58.

[3] Brasser, F., Müller, U., Dmitrienko, A., Kostiainen, K., Capkun, S., and Sadeghi, A. R. Software Grand Exposure: SGX Cache Attacks are Practical. In *11th USENIX Workshop on Offensive Technologies* (2017).

[4] Coker, G., Guttman, J., Loscocco, P., Herzog, A., Millen, J., O'Hanlon, B., and Sniffen, B. Principles of Remote Attestation. *International Journal of Information Security 10*, 2 (2011), 63–81.

[5] Costan, V., and Devadas, S. Intel SGX Explained. *IACR Cryptol 86* (2016), 1–118.

[6] Gray, J. Implementing TPM Commands in the Copland Remote Attestation Language.

[7] Guttman, J. D., and Ramsdell, J. D. Understanding attestation: Analyzing protocols that use quotes. In *Security and Trust Management - 15th International Workshop, STM 2019, Luxembourg City, Luxembourg, September 26-27, 2019, Proceedings* (2019), S. Mauw and M. Conti, Eds., vol. 11738 of *Lecture Notes in Computer Science*, Springer, pp. 89–106.

[8] Helble, S. C., Kretz, I. D., Loscocco, P. A., Ramsdell, J. D., Rowe, P. D., and Alexander, P. Flexible mechanisms for remote attestation. *ACM Trans. Priv. Secur. 24*, 4 (2021), 29:1–29:23.

[9] Jurgensen, G., Neises, M., and Alexander, P. An sel4-based architecture for layered attestation. In *Proceedings of the 7th Annual Symposium on Hot Topics in the Science of Security, HotSoS 2020, Lawrence, Kansas, USA, September 22-24, 2020* (2020), P. Alexander, D. Davidson, and B. Choi, Eds., ACM, pp. 18:1–18:2.

[10] Kretz, I. D., Ramsdell, J. D., and Rowe, P. D. Use Cases for Remote Attestation. `https://ku-sldg.github.io/copland/resources/tutorial/README`, 2008. [Online; accessed 24-March-2022].

[11] Marston, S., Li, Z., Bandyopadhyay, S., Zhang, J., and Ghalsasi, A. Cloud Computing—The business Perspective. *Decision Support Systems 51*, 1 (2011), 176–189.

[12] Mofrad, S., Zhang, F., Lu, S., and Shi, W. A comparison study of intel SGX and AMD memory encryption technology. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy, HASP@ISCA 2018, Los Angeles, CA, USA, June 02-02, 2018* (2018), J. Szefer, W. Shi, and R. B. Lee, Eds., ACM, pp. 9:1–9:8.

[13] Petz, A., and Alexander, P. A Copland Attestation Manager. In *Proceedings of the 6th Annual Symposium on Hot Topics in the Science of Security* (2019, April), pp. 1–10.

[14] Petz, A., and Alexander, P. An Infrastructure for Faithful Execution of Remote Attestation Protocols. In *NASA Formal Methods Symposium* (2021, May), Springer, Cham, pp. 268–286.

[15] Petz, A., Jurgensen, G., and Alexander, P. Design and Formal Verification of a Copland-based Attestation Protocol. In *ACM-IEEE International Conference on Formal Methods and Models for System* (2021).

[16] Ramsdell, J. D., Rowe, P. D., Alexander, P., Helble, S. C., Loscocco, P., Pendergrass, J. A., and Petz, A. Orchestrating Layered Attestations. *International Conference on Principles of Security and Trust* (2019, April), 197–221.

[17] Rowe, P. D. Bundling evidence for layered attestation. In *Trust and Trustworthy Computing - 9th International Conference, TRUST 2016, Vienna, Austria, August 29-30, 2016, Proceedings* (2016), M. Franz and P. Papadimitratos, Eds., vol. 9824 of *Lecture Notes in Computer Science*, Springer, pp. 119–139.

[18] Rowe, P. D. Confining adversary actions via measurement. In *Graphical Models for Security - Third International Workshop, GraMSec 2016, Lisbon, Portugal, June 27, 2016, Revised Selected Papers* (2016), B. Kordy, M. Ekstedt, and D. S. Kim, Eds., vol. 9987 of *Lecture Notes in Computer Science*, Springer, pp. 150–166.

[19] Rowe, P. D. Principles of layered attestation. *CoRR abs/1603.01244* (2016).

[20] Rowe, P. D., Ramsdell, J. D., and Kretz, I. D. Automated Trust Analysis of Copland Specifications for Layered Attestations. In *23rd International Symposium on Principles and Practice of Declarative Programming* (2021, September), pp. 1–15.

[21] Saghafi, S., Danas, R., and Dougherty, D. J. Exploring theories with a model-finding assistant. In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings* (2015), A. P. Felty and A. Middeldorp, Eds., vol. 9195 of *Lecture Notes in Computer Science*, Springer, pp. 434–449.

[22] Sardar, M. U., Faqeh, R., and Fetzer, C. Formal foundations for intel SGX data center attestation primitives. In *Formal Methods and Software Engineering - 22nd International Conference on Formal Engineering Methods, ICFEM 2020, Singapore, Singapore, March 1-3, 2021, Proceedings* (2020), S. Lin, Z. Hou, and B. P. Mahony, Eds., vol. 12531 of *Lecture Notes in Computer Science*, Springer, pp. 268–283.

[23] Swami, Y. Intel SGX Remote Attestation is not Sufficient. In *Procedings Black Hat USA* (2017, July).

[24] Tomescu, A. SGX and Haven. `http://people.csail.mit.edu/alinush/6.858-fall-2014/2015/l08-sgx.html`, 2015. [Online; accessed 9-April-2022].

[25] van Schaik, S., Kwong, A., Genkin, D., and Yarom, Y. SGAxe: How SGX fails in Practice.

[26] van Schaik, S., Minkin, M., Kwong, A., Genkin, D., and Yarom, Y. CacheOut: Leaking Data on Intel CPUs via Cache Evictions. In *2021 IEEE Symposium on Security and Privacy (SP)* (2021, May), IEEE, pp. 339–354.

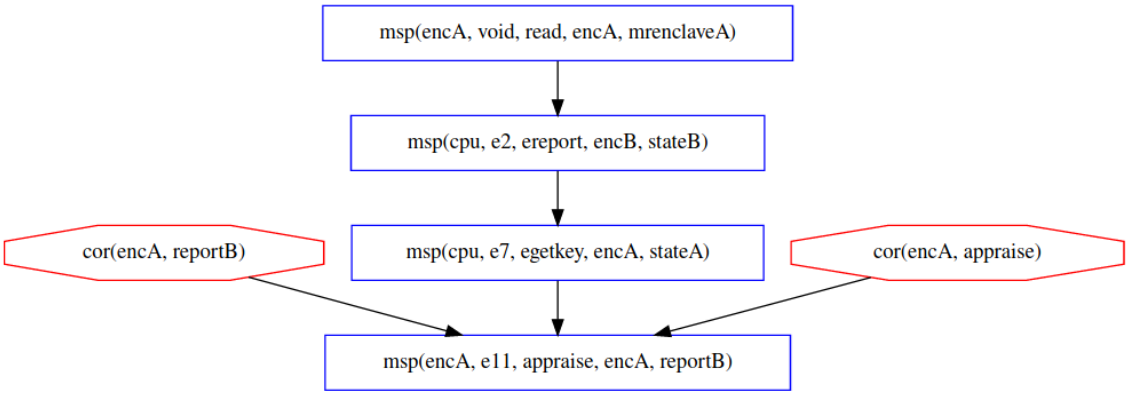
[27] Vill, H. SGX Attestation Process.

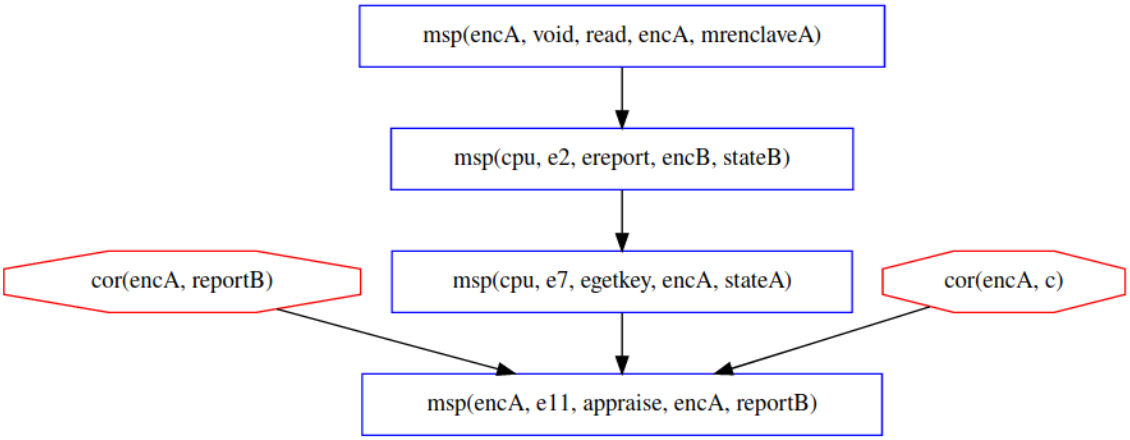
[28] Wang, W., Chen, G., Pan, X., Zhang, Y., Wang, X., Bindschaedler, V., and Gunter, C. A. Leaky Cauldron on the Dark Land: Understanding Memory Side-channel Hazards in SGX. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017, October), pp. 2421–2434.

Appendix A

Basic local attestation scenarios

The following figures show the attack models obtained by Chase after applying the basic assumptions discussed in Chapter 5.1 to the simplified local attestation Copland phrase.

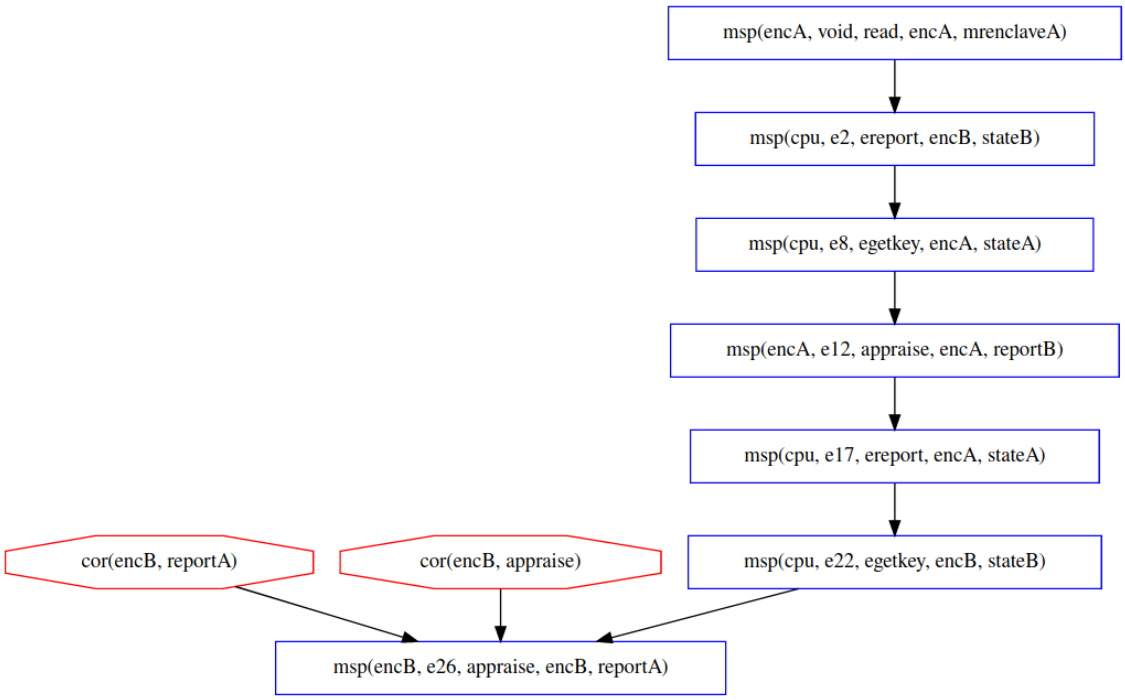


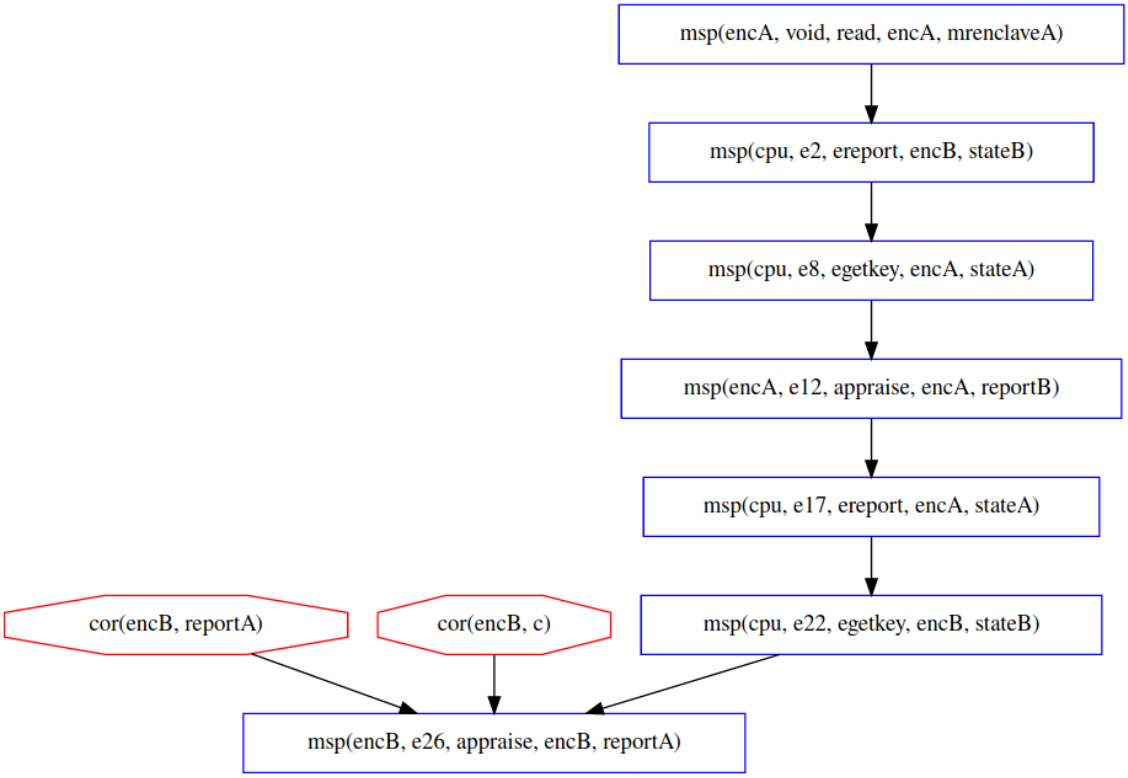

msp(encA, void, read, encA, mrenclaveA)
msp(cpu, e2, ereport, encB, stateB)
cor(encA, reportB)
msp(cpu, e7, egetkey, encA, stateA)
cor(encA, c)
msp(encA, e11, appraise, encA, reportB)

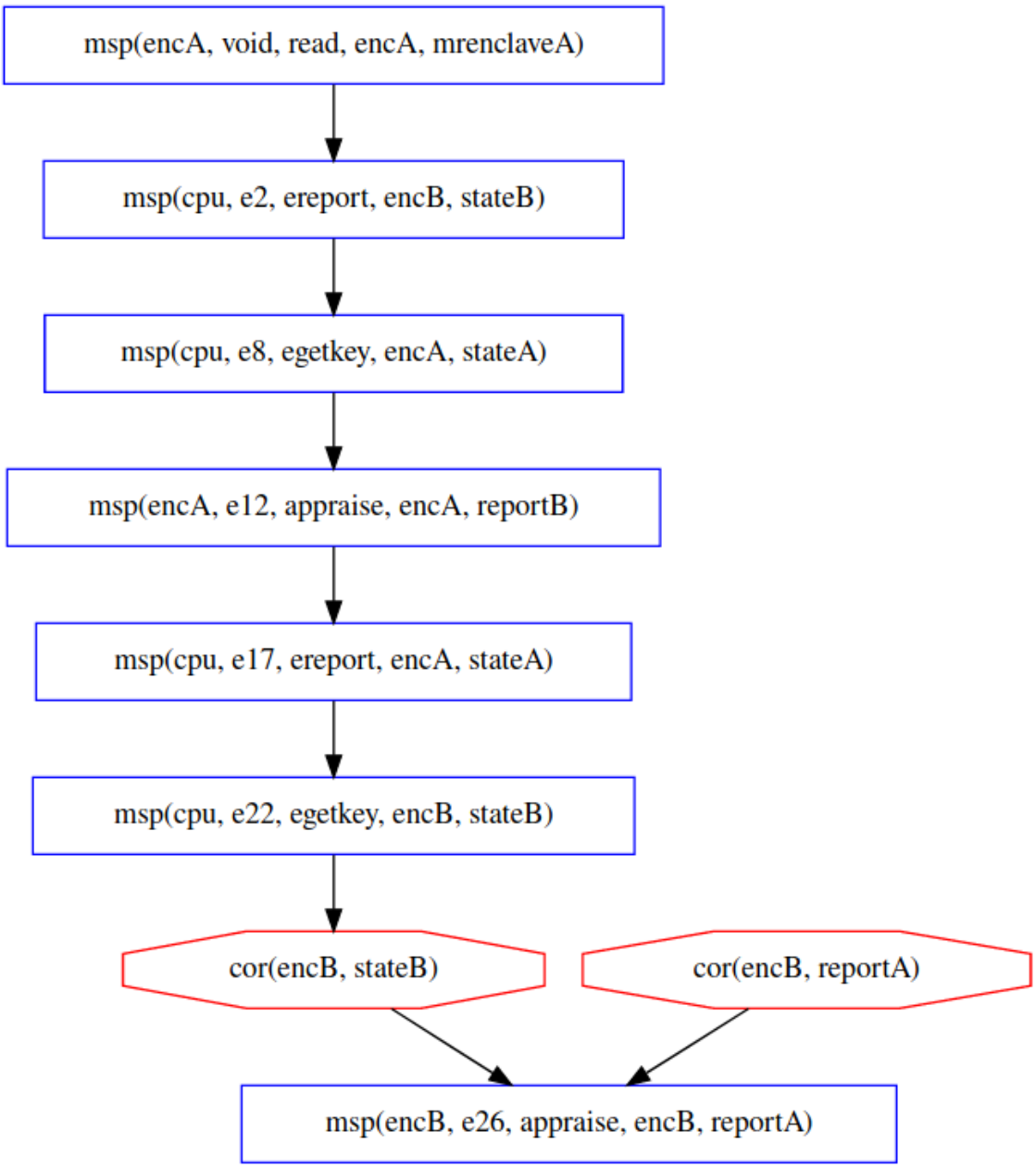
Appendix B

Local attestation scenarios

The following figures show the attack models obtained by Chase after applying the basic assumptions discussed in Chapter 5.1 to the complete local attestation Copland phrase.



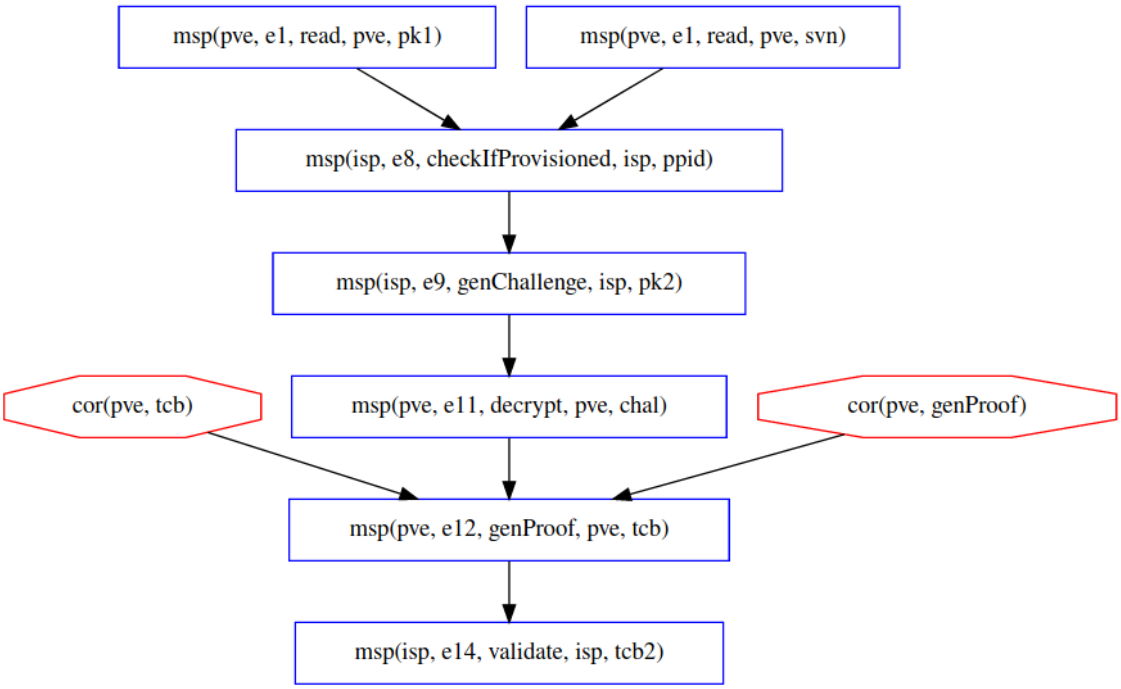

msp(encA, void, read, encA, mrenclaveA)
msp(cpu, e2, ereport, encB, stateB)
msp(cpu, e8, egetkey, encA, stateA)
msp(encA, e12, appraise, encA, reportB)
msp(cpu, e17, ereport, encA, stateA)
msp(cpu, e22, egetkey, encB, stateB)
cor(encB, reportA)
cor(encB, c)
msp(encB, e26, appraise, encB, reportA)

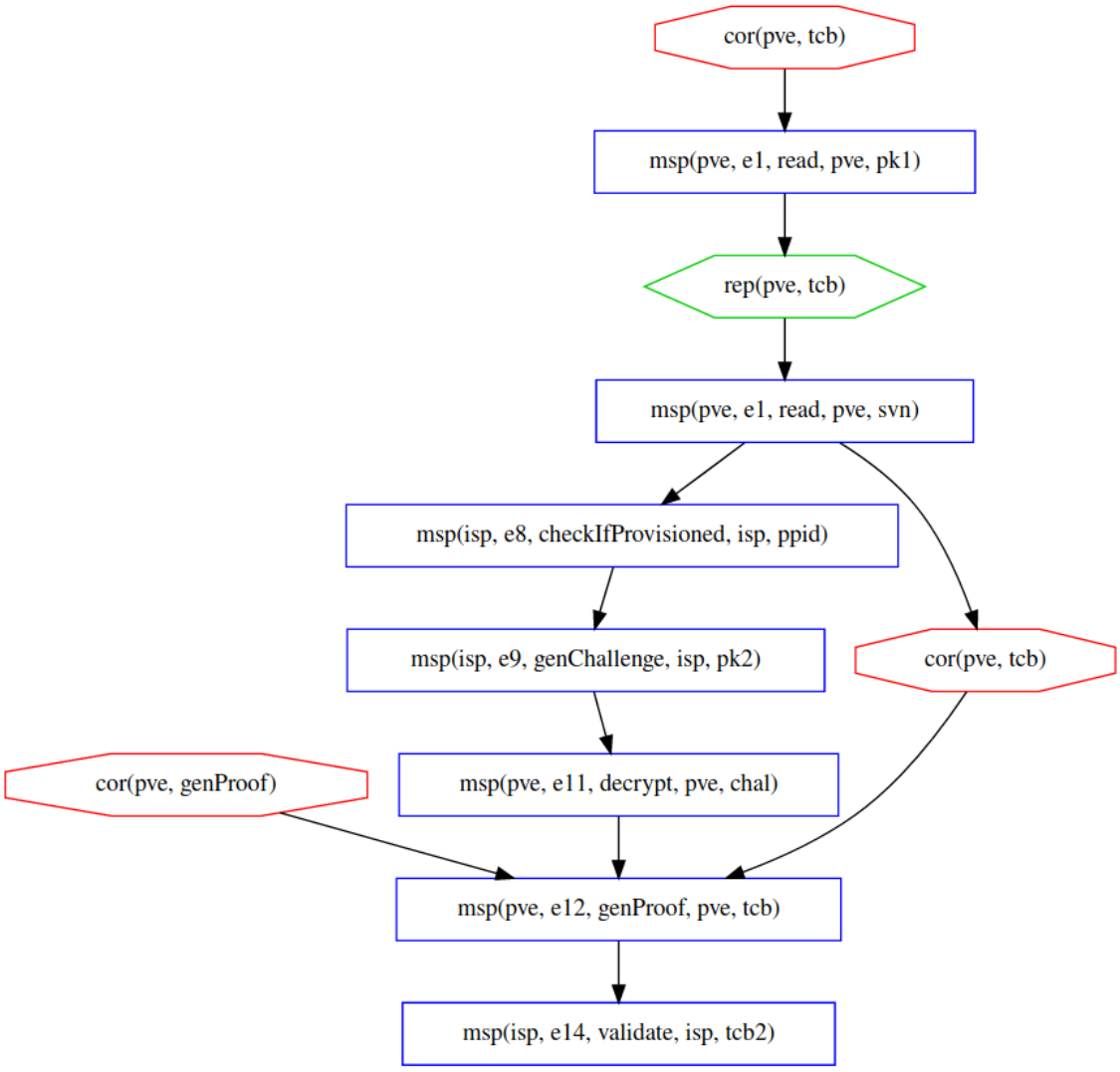

msp(encA, void, read, encA, mrenclaveA)
msp(cpu, e2, ereport, encB, stateB)
msp(cpu, e8, egetkey, encA, stateA)
msp(encA, e12, appraise, encA, reportB)
msp(cpu, e17, ereport, encA, stateA)
msp(cpu, e22, egetkey, encB, stateB)
cor(encB, stateB)
cor(encB, reportA)
msp(encB, e26, appraise, encB, reportA)

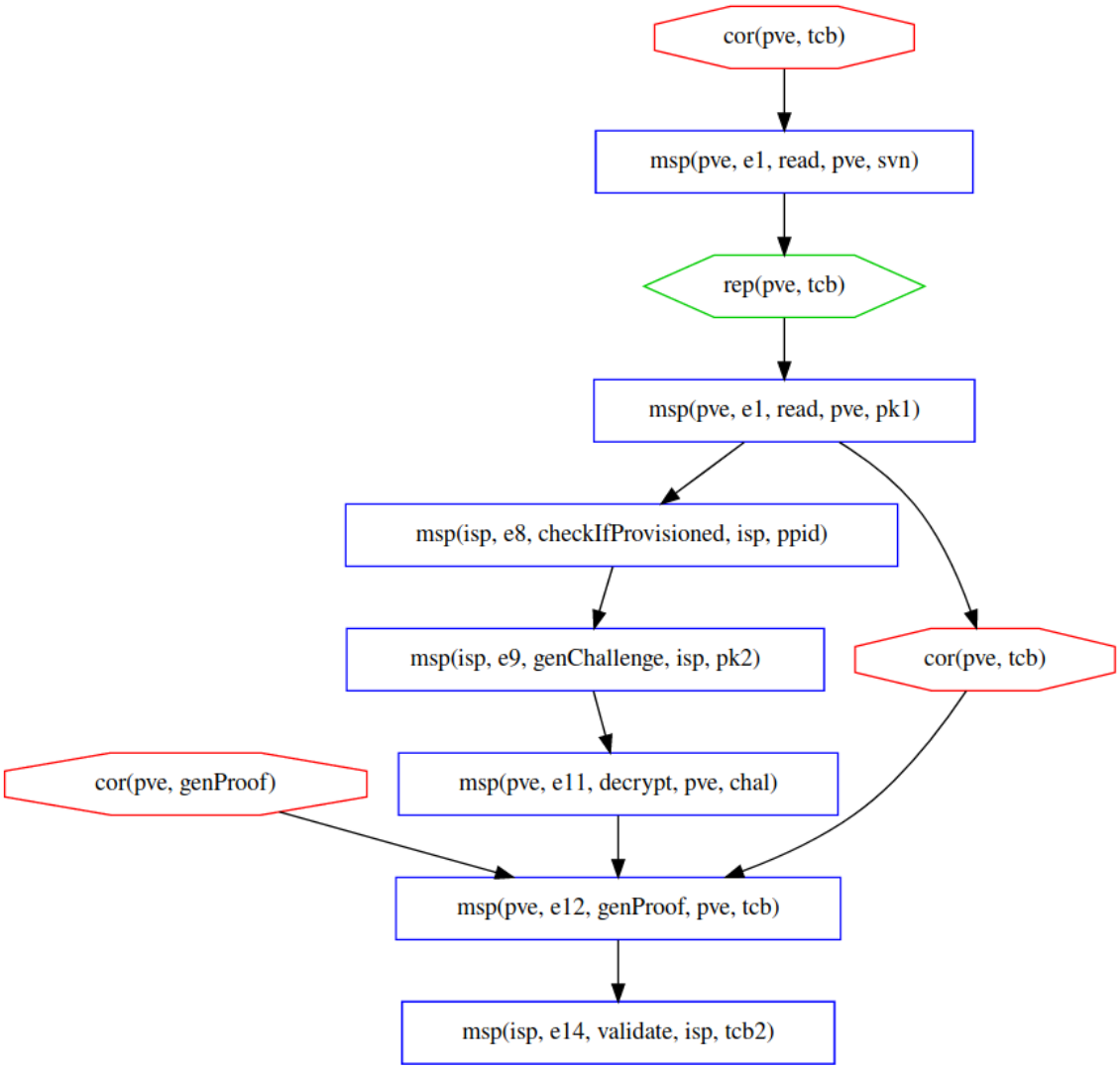
Appendix C

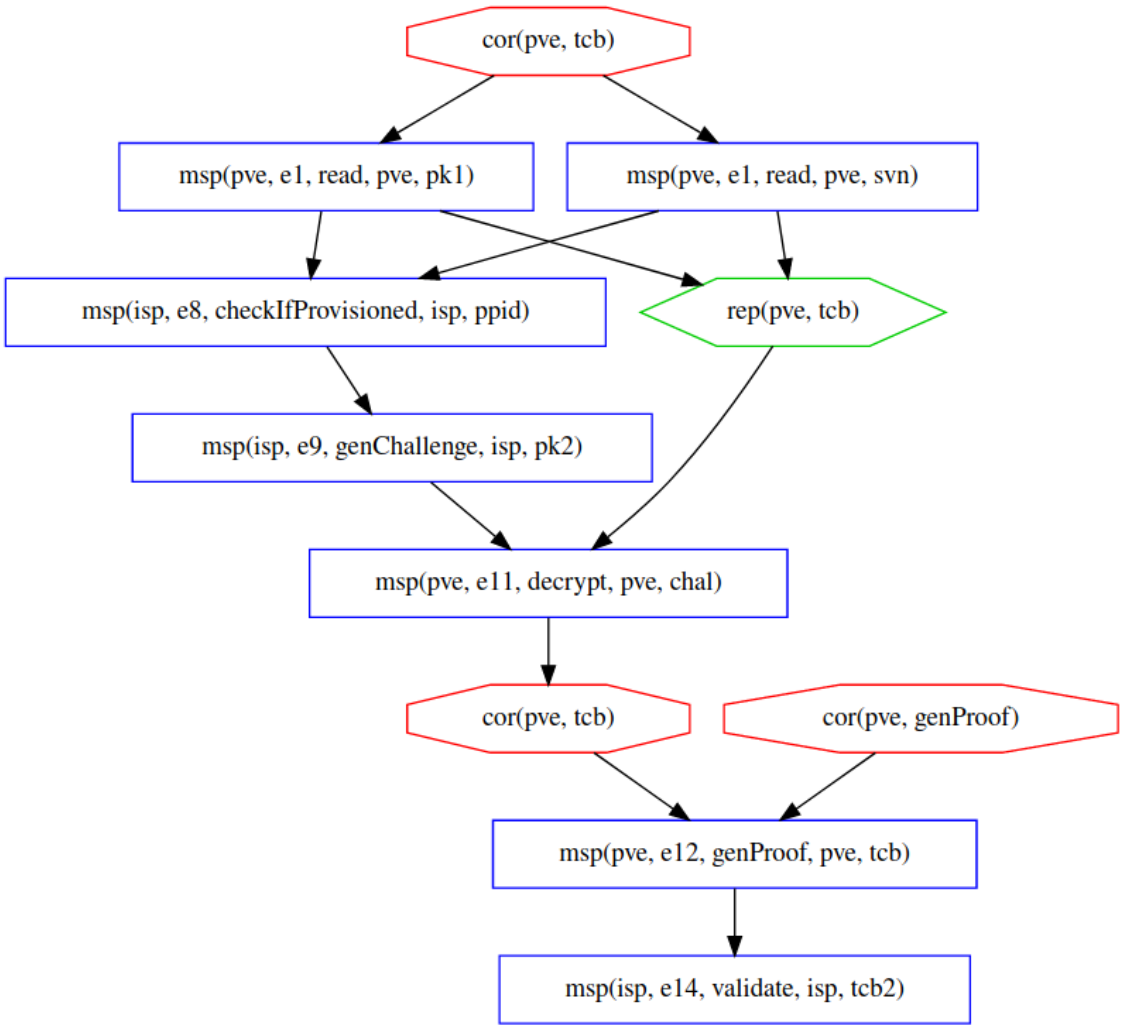
Platform Provisioning scenarios

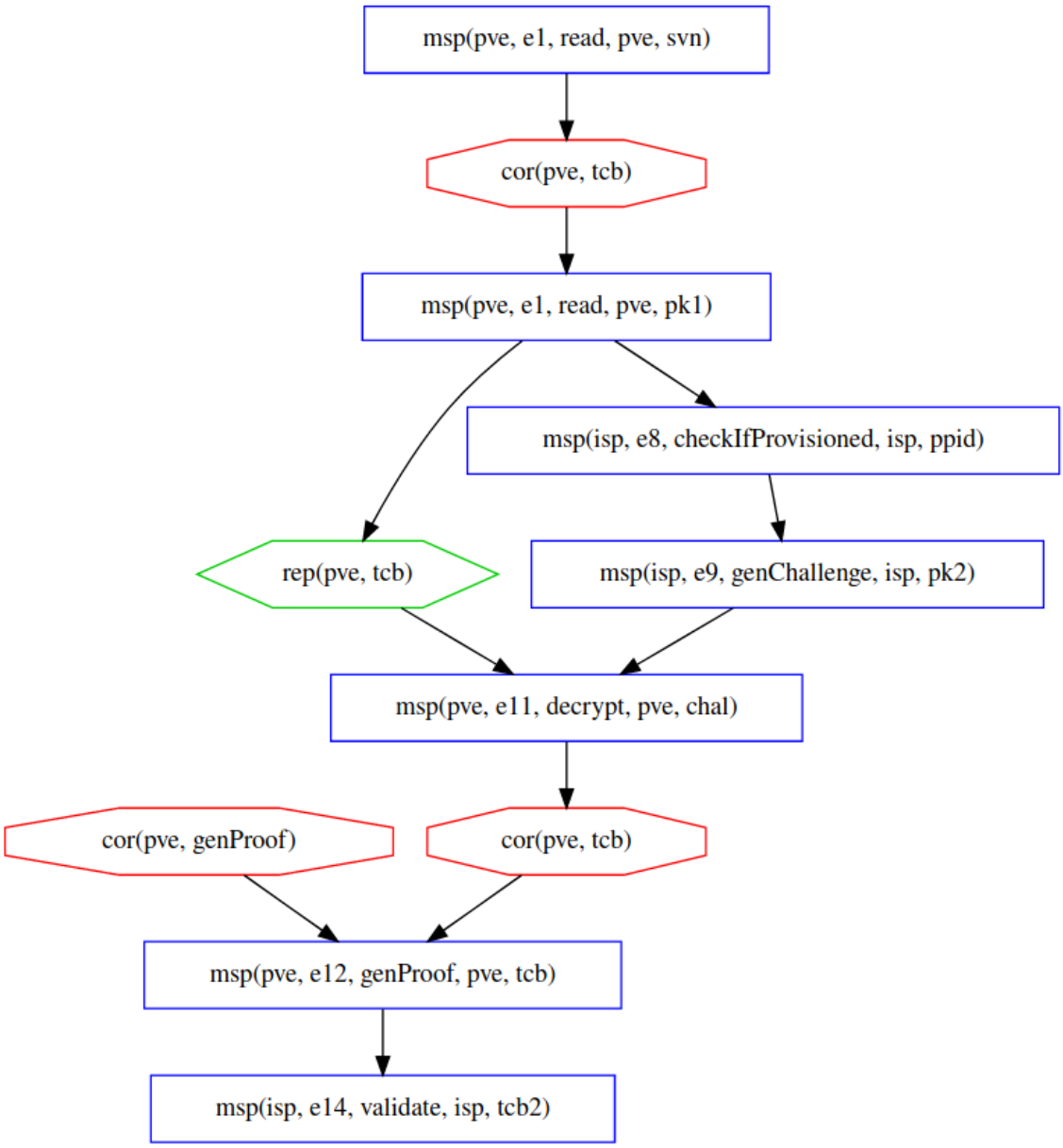
The following figures show the attack models obtained by Chase after applying the basic assumptions discussed in Chapter 5.2 to the Platform Provisioning Copland phrase.

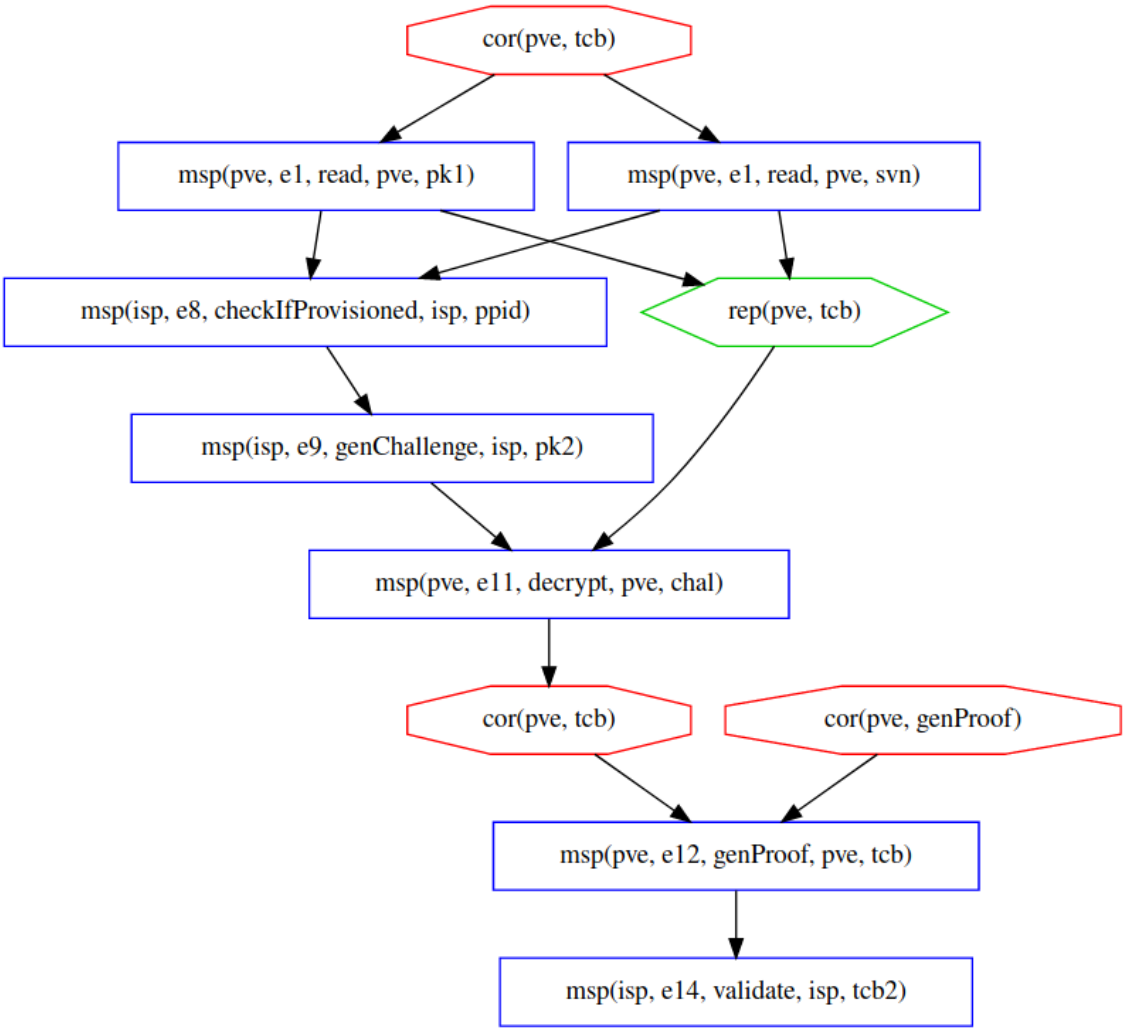


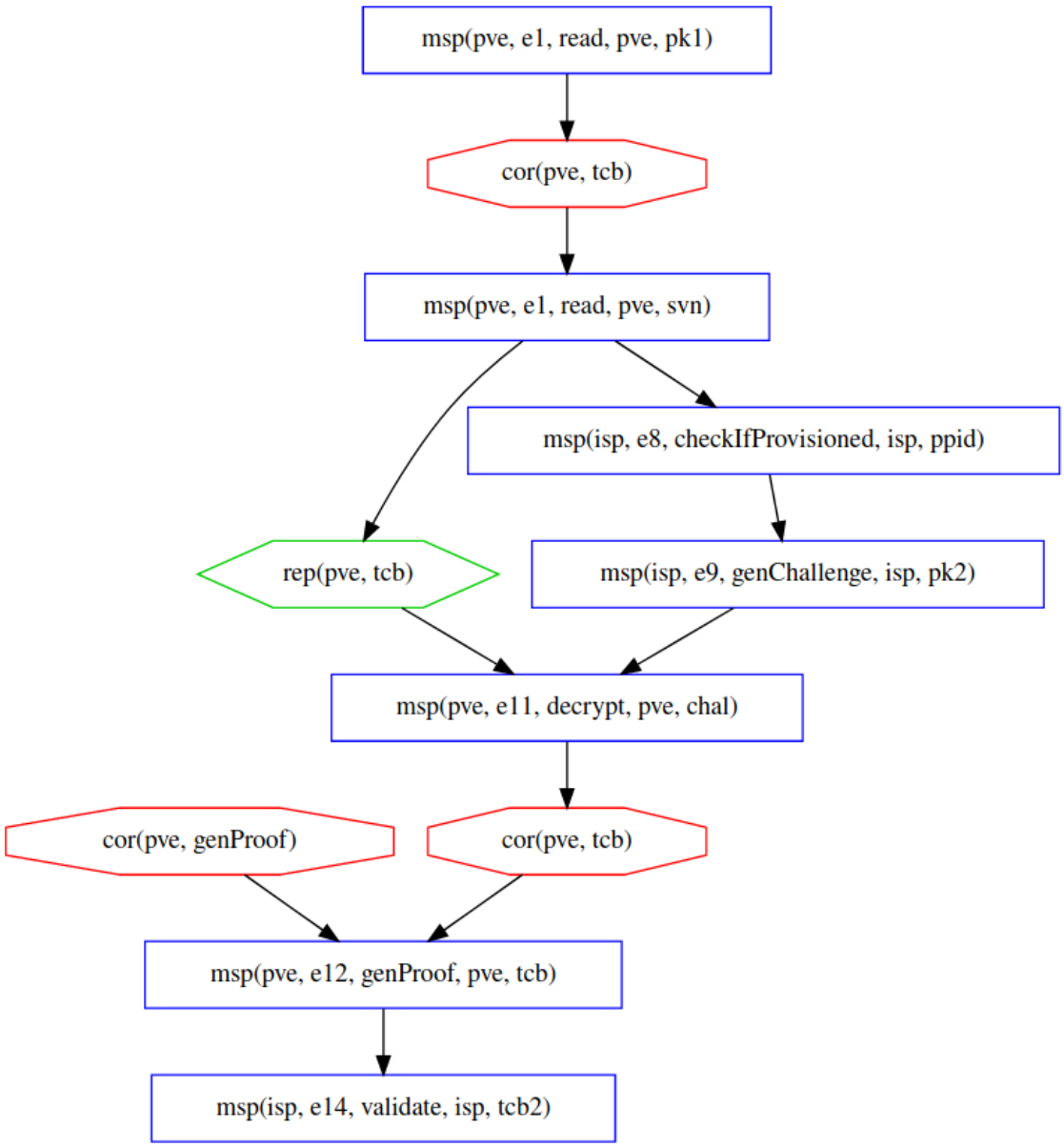

cor(pve, tcb)
msp(pve, e1, read, pve, pk1)
rep(pve, tcb)
msp(pve, e1, read, pve, svn)
msp(isp, e8, checkIfProvisioned, isp, ppid)
msp(isp, e9, genChallenge, isp, pk2)
cor(pve, tcb)
cor(pve, genProof)
msp(pve, e11, decrypt, pve, chal)
msp(pve, e12, genProof, pve, tcb)
msp(isp, e14, validate, isp, tcb2)

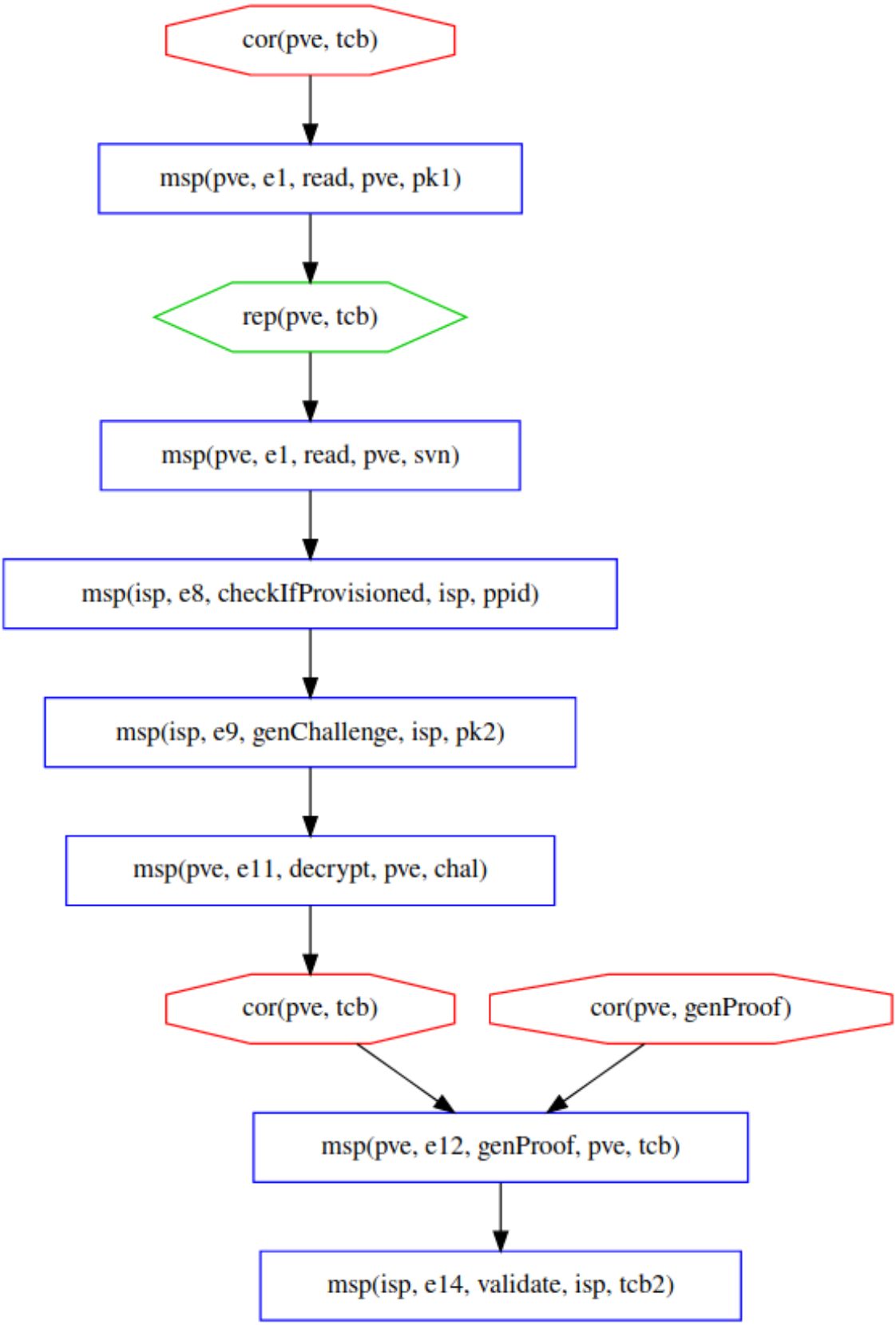

cor(pve, tcb)
msp(pve, e1, read, pve, svn)
rep(pve, tcb)
msp(pve, e1, read, pve, pk1)
msp(isp, e8, checkIfProvisioned, isp, ppid)
msp(isp, e9, genChallenge, isp, pk2)
cor(pve, tcb)
cor(pve, genProof)
msp(pve, e11, decrypt, pve, chal)
msp(pve, e12, genProof, pve, tcb)
msp(isp, e14, validate, isp, tcb2)

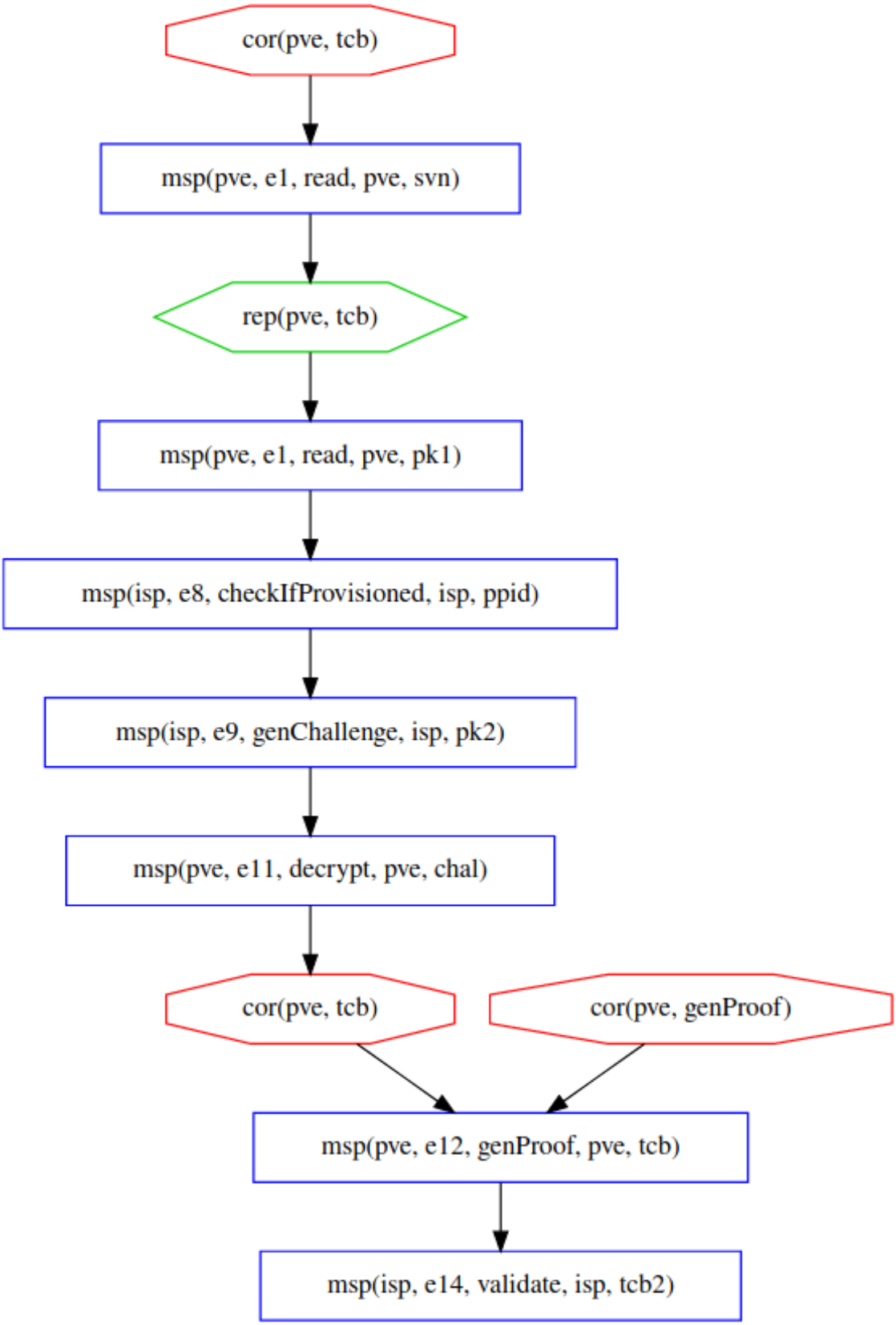

cor(pve, tcb)
msp(pve, e1, read, pve, pk1)
msp(pve, e1, read, pve, svn)
msp(isp, e8, checkIfProvisioned, isp, ppid)
rep(pve, tcb)
msp(isp, e9, genChallenge, isp, pk2)
msp(pve, e11, decrypt, pve, chal)
cor(pve, tcb)
cor(pve, genProof)
msp(pve, e12, genProof, pve, tcb)
msp(isp, e14, validate, isp, tcb2)


msp(pve, e1, read, pve, svn)
cor(pve, tcb)
msp(pve, e1, read, pve, pk1)
msp(isp, e8, checkIfProvisioned, isp, ppid)
rep(pve, tcb)
msp(isp, e9, genChallenge, isp, pk2)
msp(pve, e11, decrypt, pve, chal)
cor(pve, genProof)
cor(pve, tcb)
msp(pve, e12, genProof, pve, tcb)
msp(isp, e14, validate, isp, tcb2)


cor(pve, tcb)
msp(pve, e1, read, pve, pk1)
msp(pve, e1, read, pve, svn)
msp(isp, e8, checkIfProvisioned, isp, ppid)
rep(pve, tcb)
msp(isp, e9, genChallenge, isp, pk2)
msp(pve, e11, decrypt, pve, chal)
cor(pve, tcb)
cor(pve, genProof)
msp(pve, e12, genProof, pve, tcb)
msp(isp, e14, validate, isp, tcb2)


msp(pve, e1, read, pve, pk1)
cor(pve, tcb)
msp(pve, e1, read, pve, svn)
msp(isp, e8, checkIfProvisioned, isp, ppid)
rep(pve, tcb)
msp(isp, e9, genChallenge, isp, pk2)
msp(pve, e11, decrypt, pve, chal)
cor(pve, genProof)
cor(pve, tcb)
msp(pve, e12, genProof, pve, tcb)
msp(isp, e14, validate, isp, tcb2)

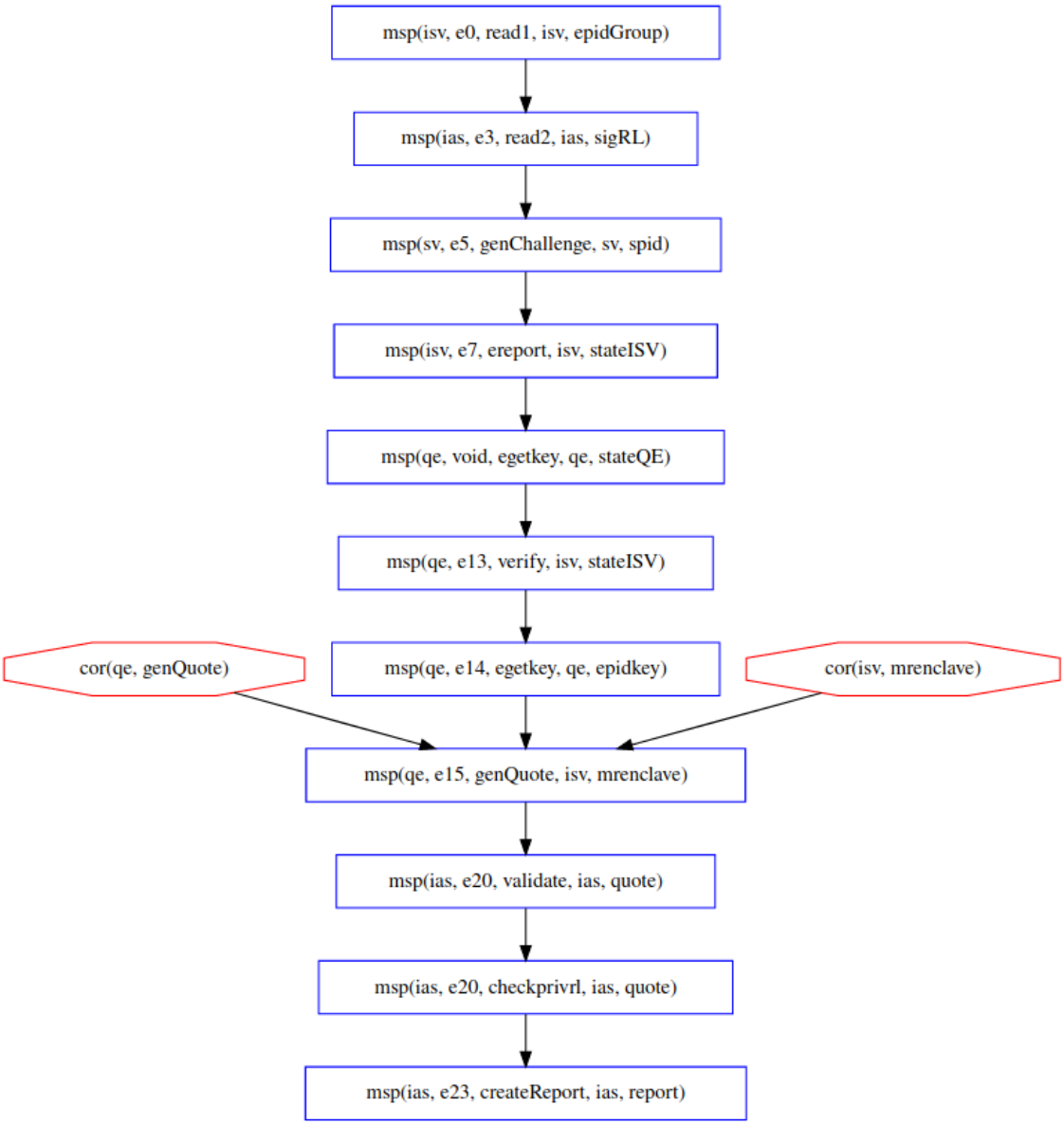

cor(pve, tcb)
msp(pve, e1, read, pve, pk1)
rep(pve, tcb)
msp(pve, e1, read, pve, svn)
msp(isp, e8, checkIfProvisioned, isp, ppid)
msp(isp, e9, genChallenge, isp, pk2)
msp(pve, e11, decrypt, pve, chal)
cor(pve, tcb)
cor(pve, genProof)
msp(pve, e12, genProof, pve, tcb)
msp(isp, e14, validate, isp, tcb2)

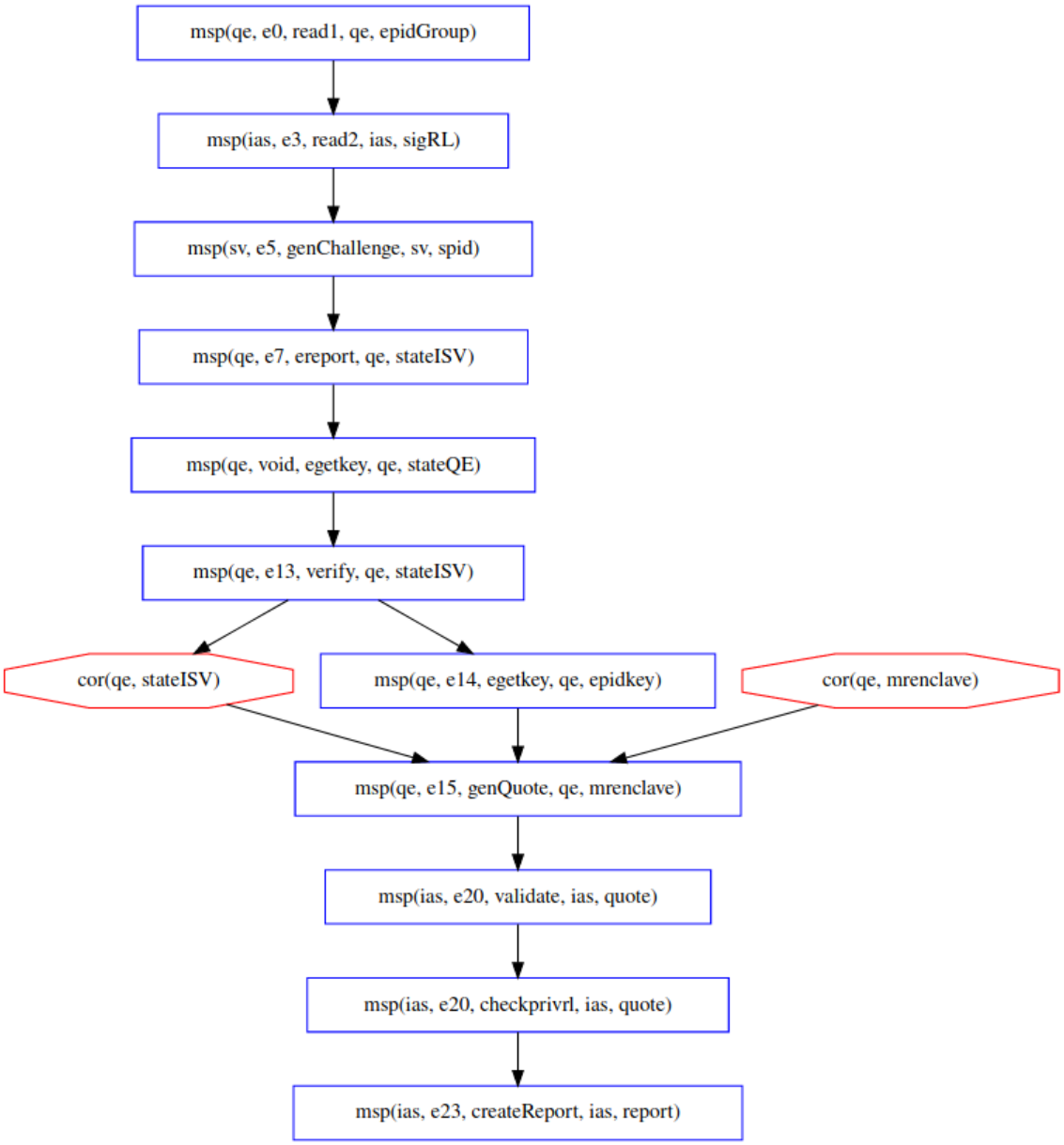

cor(pve, tcb)
msp(pve, e1, read, pve, svn)
rep(pve, tcb)
msp(pve, e1, read, pve, pk1)
msp(isp, e8, checkIfProvisioned, isp, ppid)
msp(isp, e9, genChallenge, isp, pk2)
msp(pve, e11, decrypt, pve, chal)
cor(pve, tcb)
cor(pve, genProof)
msp(pve, e12, genProof, pve, tcb)
msp(isp, e14, validate, isp, tcb2)

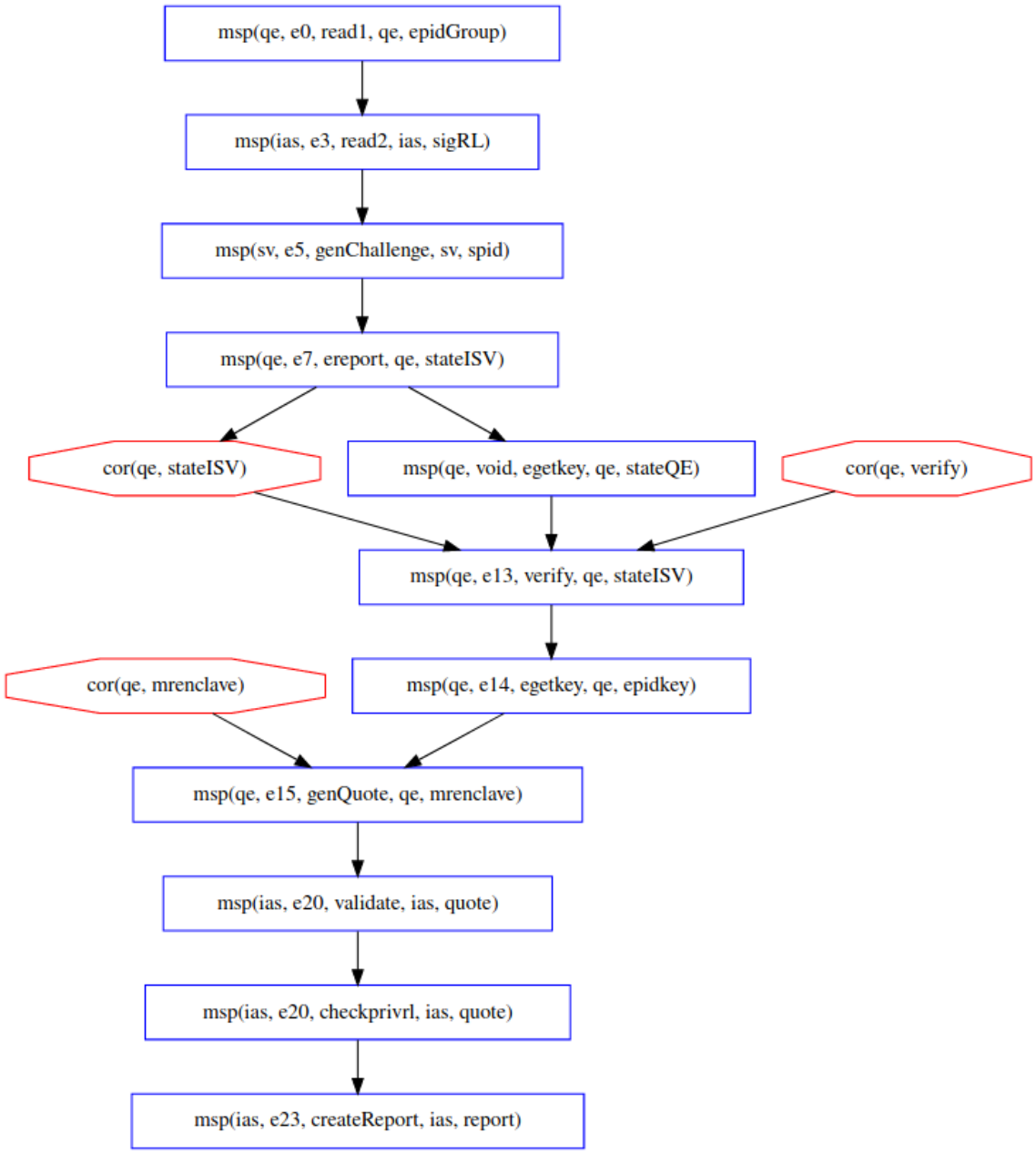
Appendix D

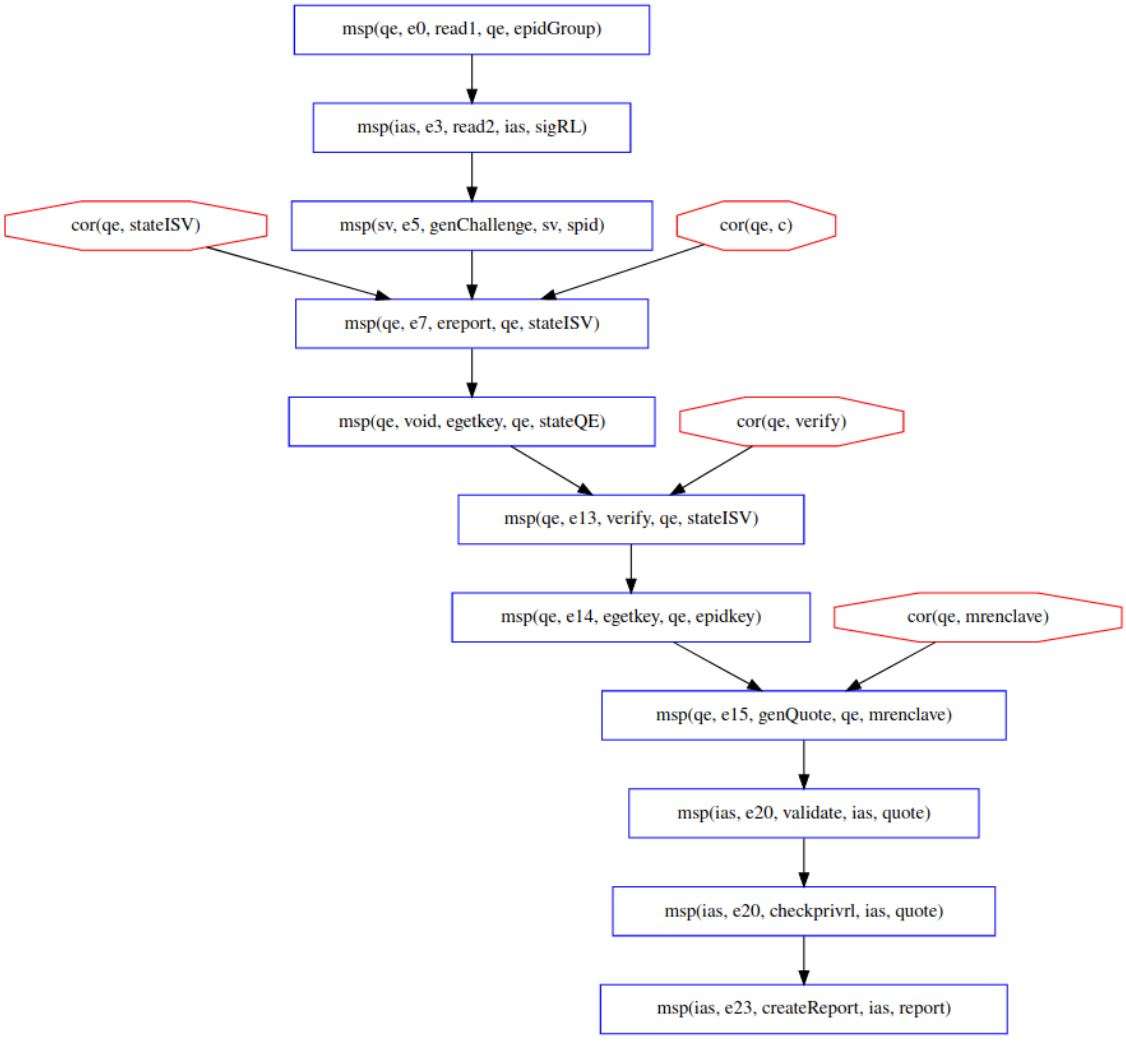
Remote attestation scenarios

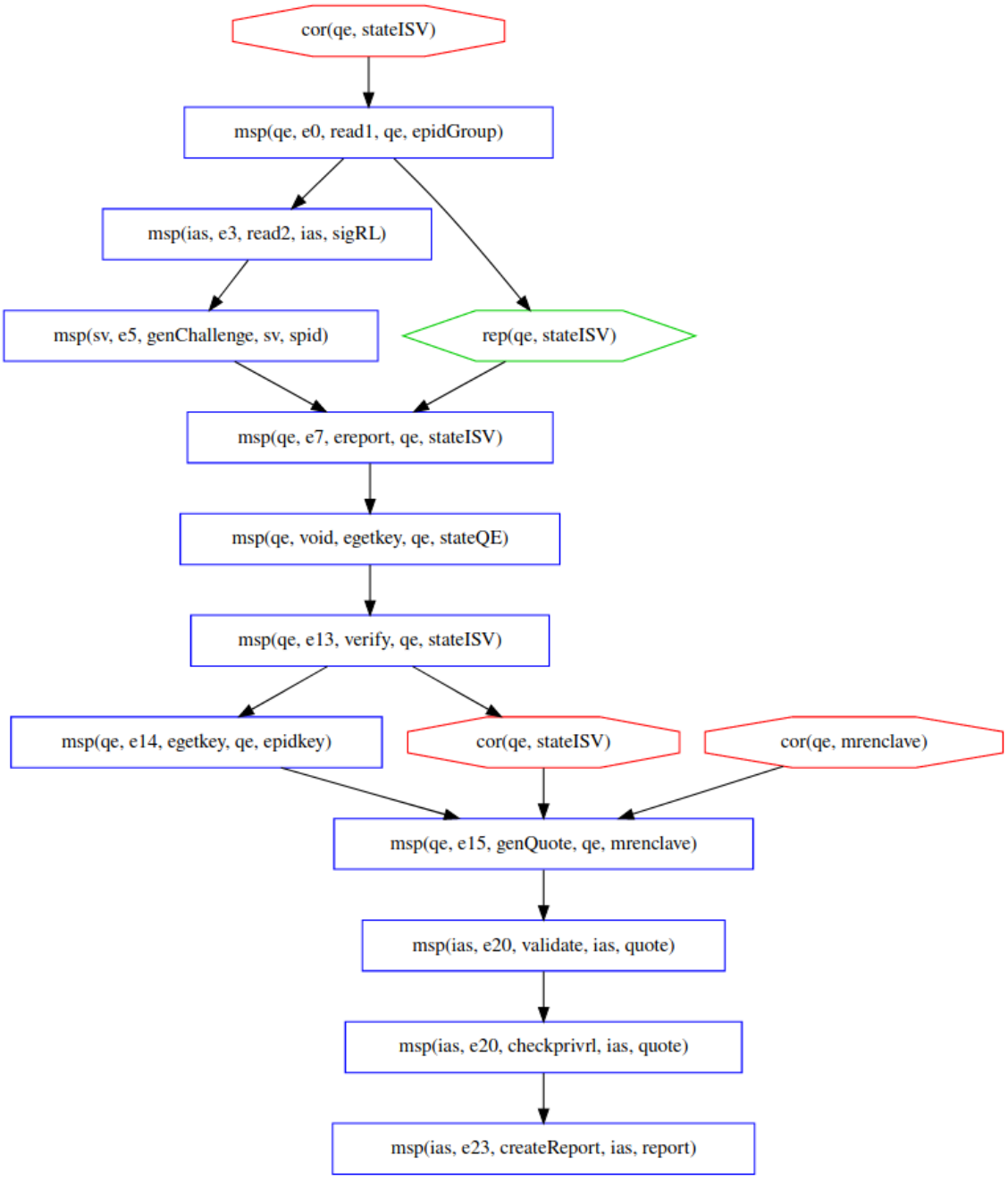
The following figures show the attack models obtained by Chase after applying the basic assumptions discussed in Chapter 5 to the remote attestation Copland phrase.

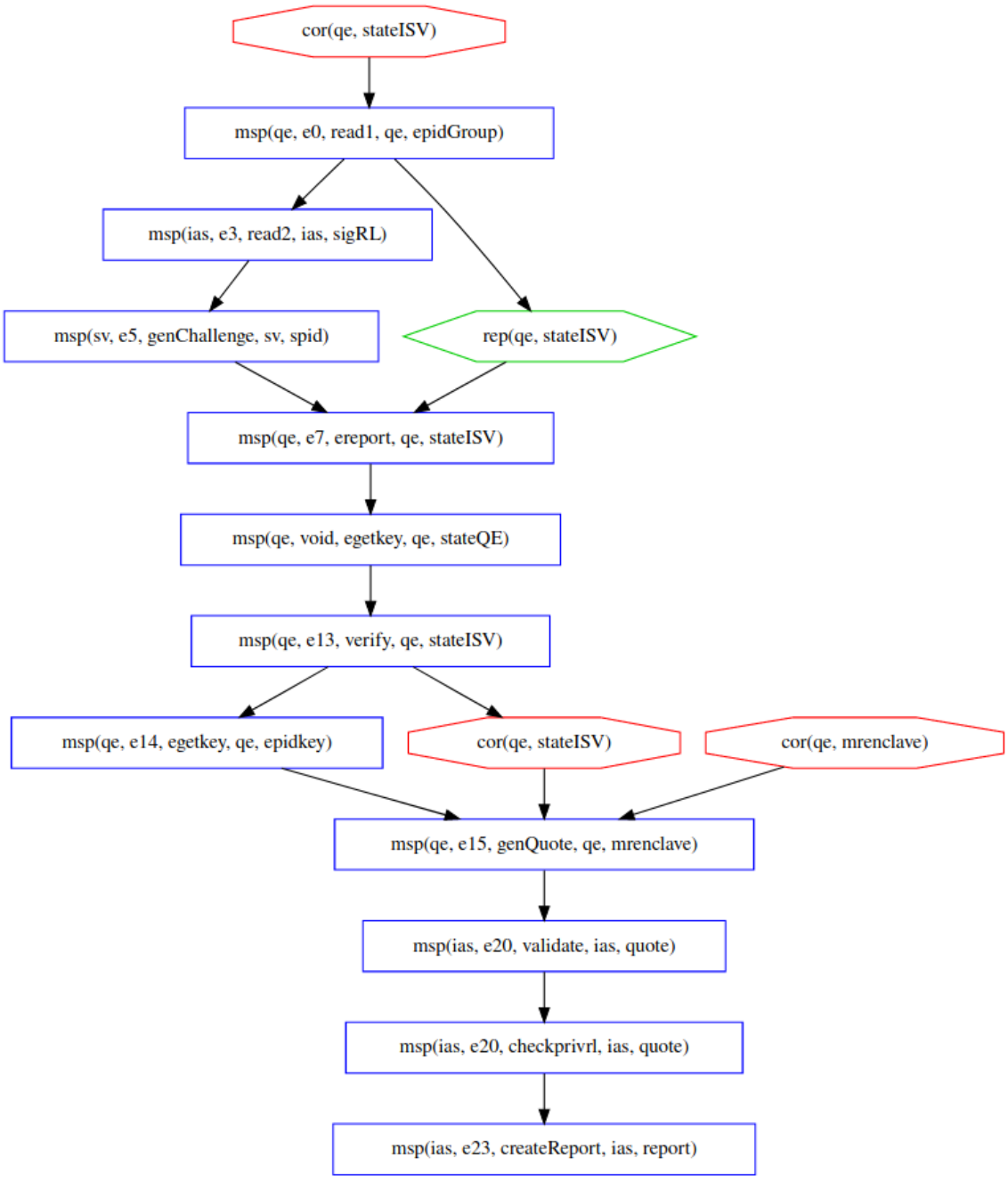

msp(isv, e0, read1, isv, epidGroup)
msp(ias, e3, read2, ias, sigRL)
msp(sv, e5, genChallenge, sv, spid)
msp(isv, e7, ereport, isv, stateISV)
msp(qe, void, egetkey, qe, stateQE)
msp(qe, e13, verify, isv, stateISV)
cor(qe, genQuote)
msp(qe, e14, egetkey, qe, epidkey)
cor(isv, mrenclave)
msp(qe, e15, genQuote, isv, mrenclave)
msp(ias, e20, validate, ias, quote)
msp(ias, e20, checkprivrl, ias, quote)
msp(ias, e23, createReport, ias, report)

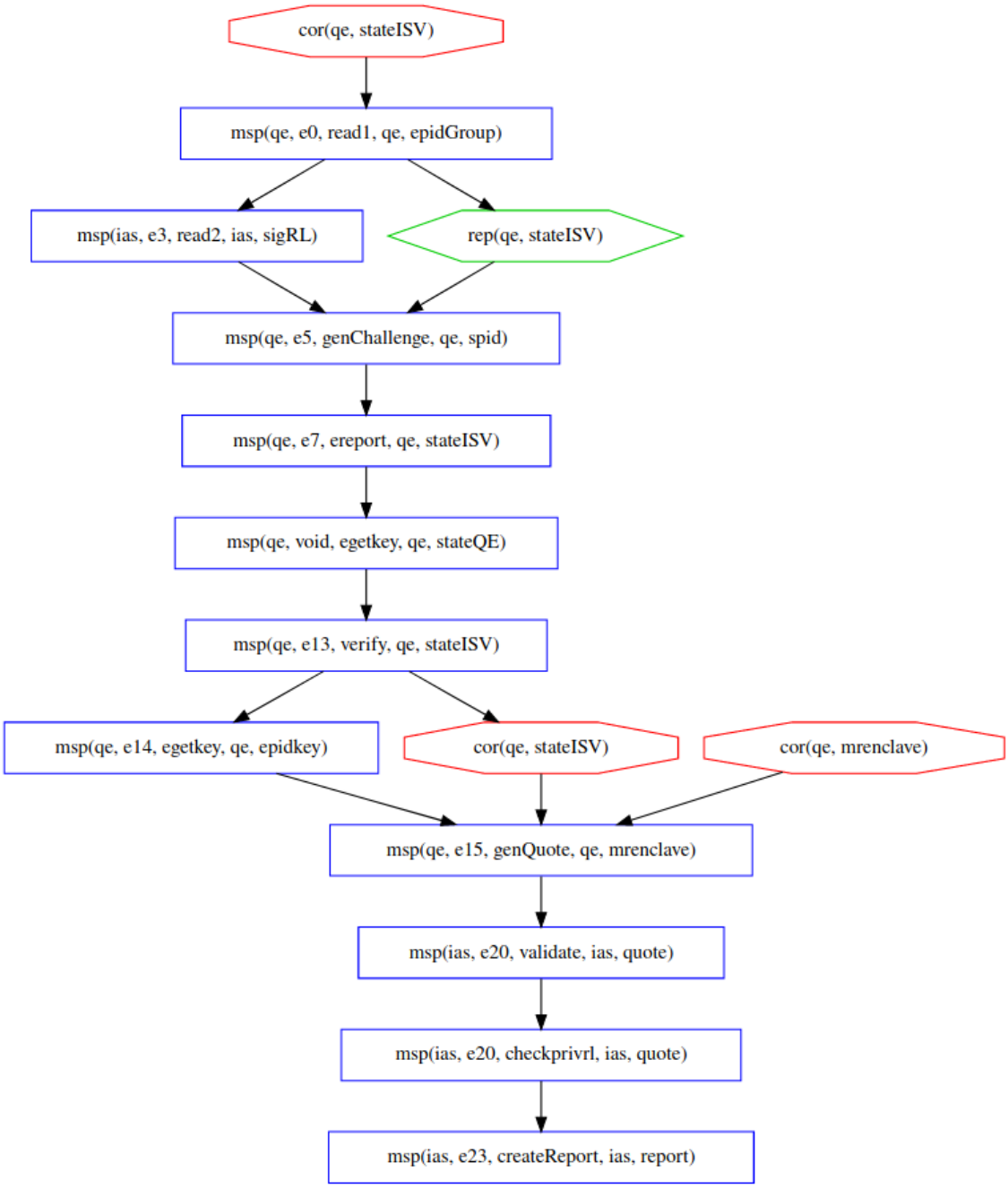

msp(qe, e0, read1, qe, epidGroup)
msp(ias, e3, read2, ias, sigRL)
msp(sv, e5, genChallenge, sv, spid)
msp(qe, e7, ereport, qe, stateISV)
msp(qe, void, egetkey, qe, stateQE)
msp(qe, e13, verify, qe, stateISV)
cor(qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(ias, e20, validate, ias, quote)
msp(ias, e20, checkprivrl, ias, quote)
msp(ias, e23, createReport, ias, report)

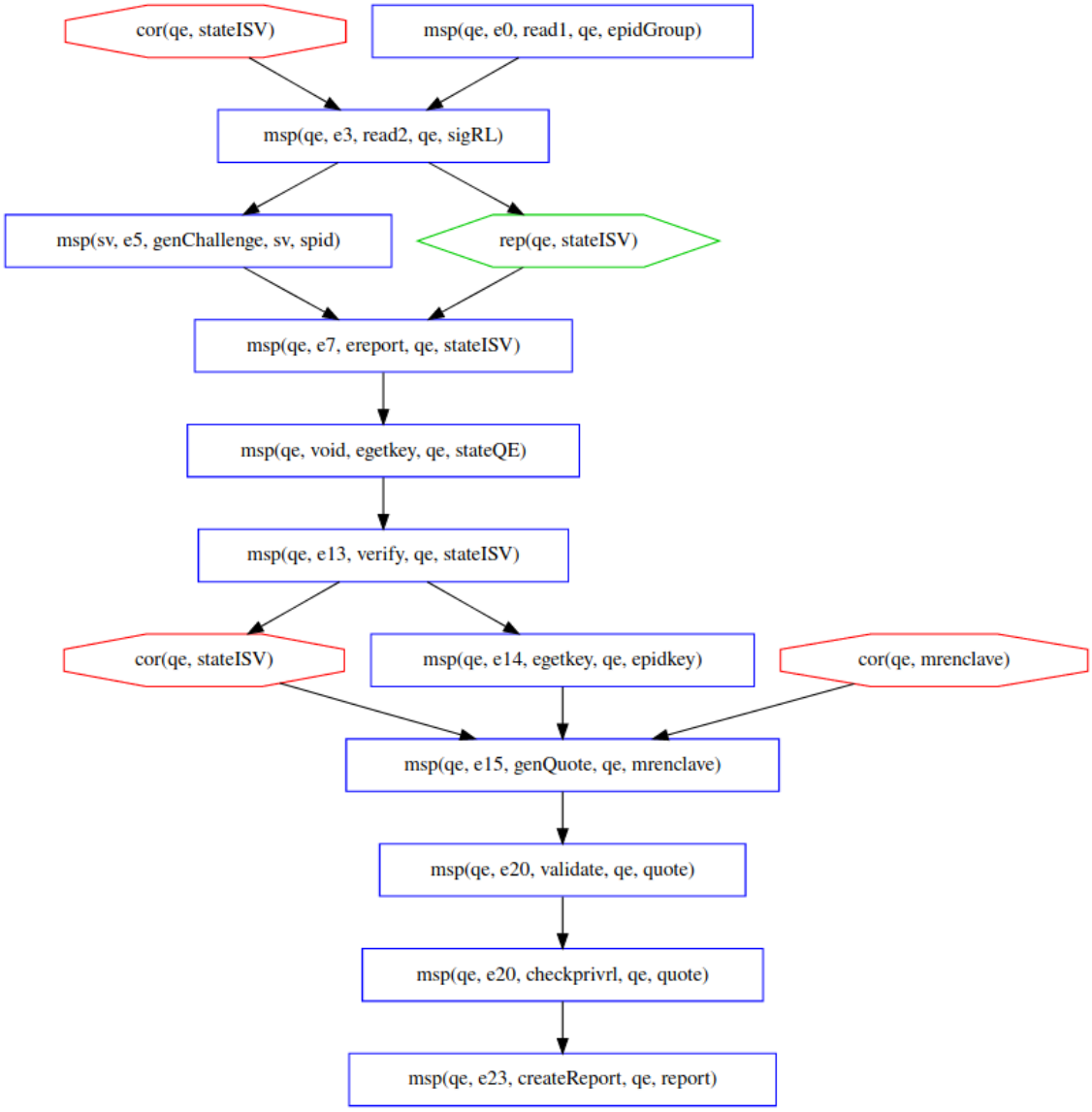

msp(qe, e0, read1, qe, epidGroup)
msp(ias, e3, read2, ias, sigRL)
msp(sv, e5, genChallenge, sv, spid)
msp(qe, e7, ereport, qe, stateISV)
cor(qe, stateISV)
msp(qe, void, egetkey, qe, stateQE)
cor(qe, verify)
msp(qe, e13, verify, qe, stateISV)
cor(qe, mrenclave)
msp(qe, e14, egetkey, qe, epidkey)
msp(qe, e15, genQuote, qe, mrenclave)
msp(ias, e20, validate, ias, quote)
msp(ias, e20, checkprivrl, ias, quote)
msp(ias, e23, createReport, ias, report)

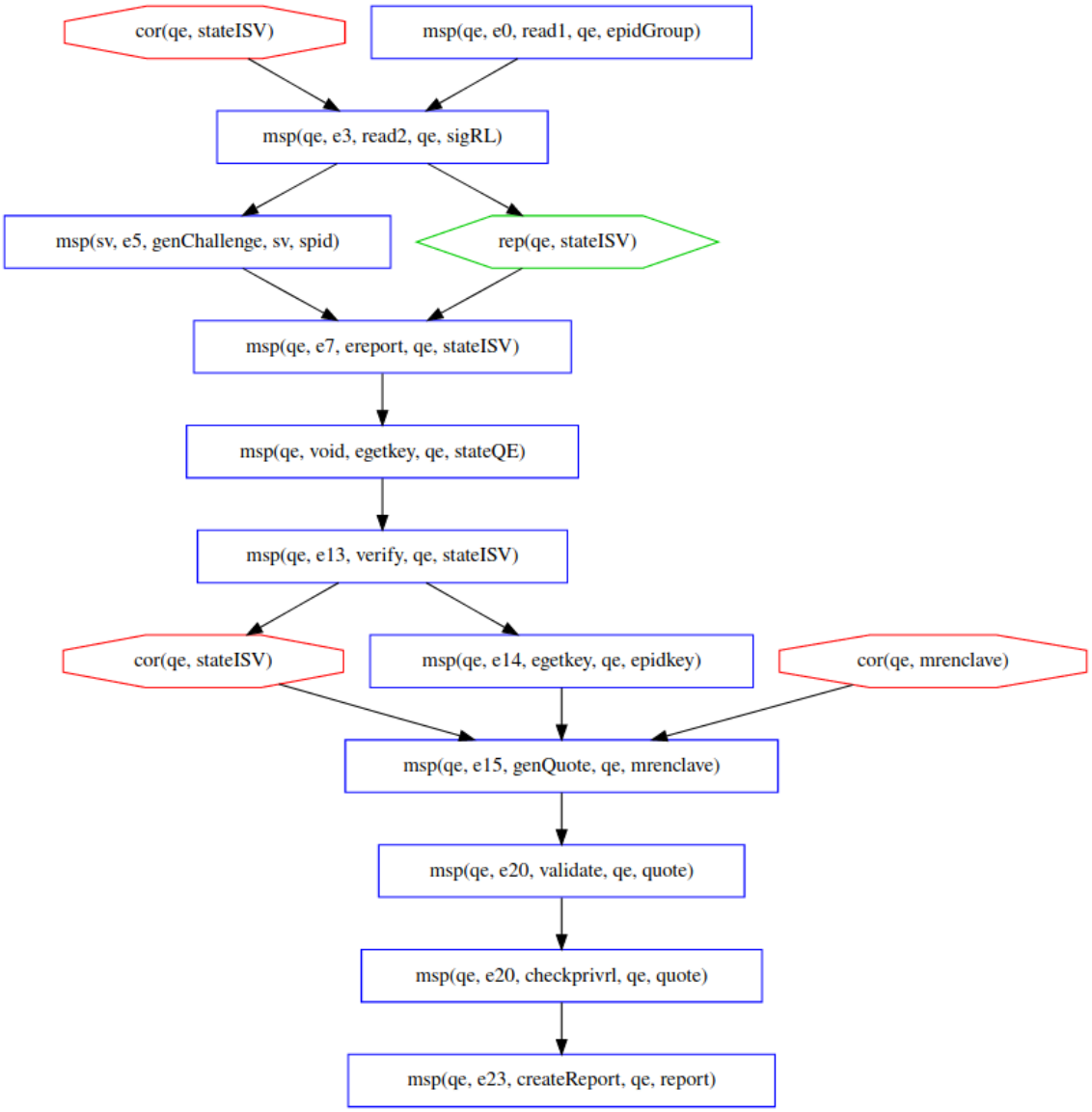

msp(qe, e0, read1, qe, epidGroup)
msp(ias, e3, read2, ias, sigRL)
cor(qe, stateISV)
msp(sv, e5, genChallenge, sv, spid)
cor(qe, c)
msp(qe, e7, ereport, qe, stateISV)
msp(qe, void, egetkey, qe, stateQE)
cor(qe, verify)
msp(qe, e13, verify, qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(ias, e20, validate, ias, quote)
msp(ias, e20, checkprivrl, ias, quote)
msp(ias, e23, createReport, ias, report)

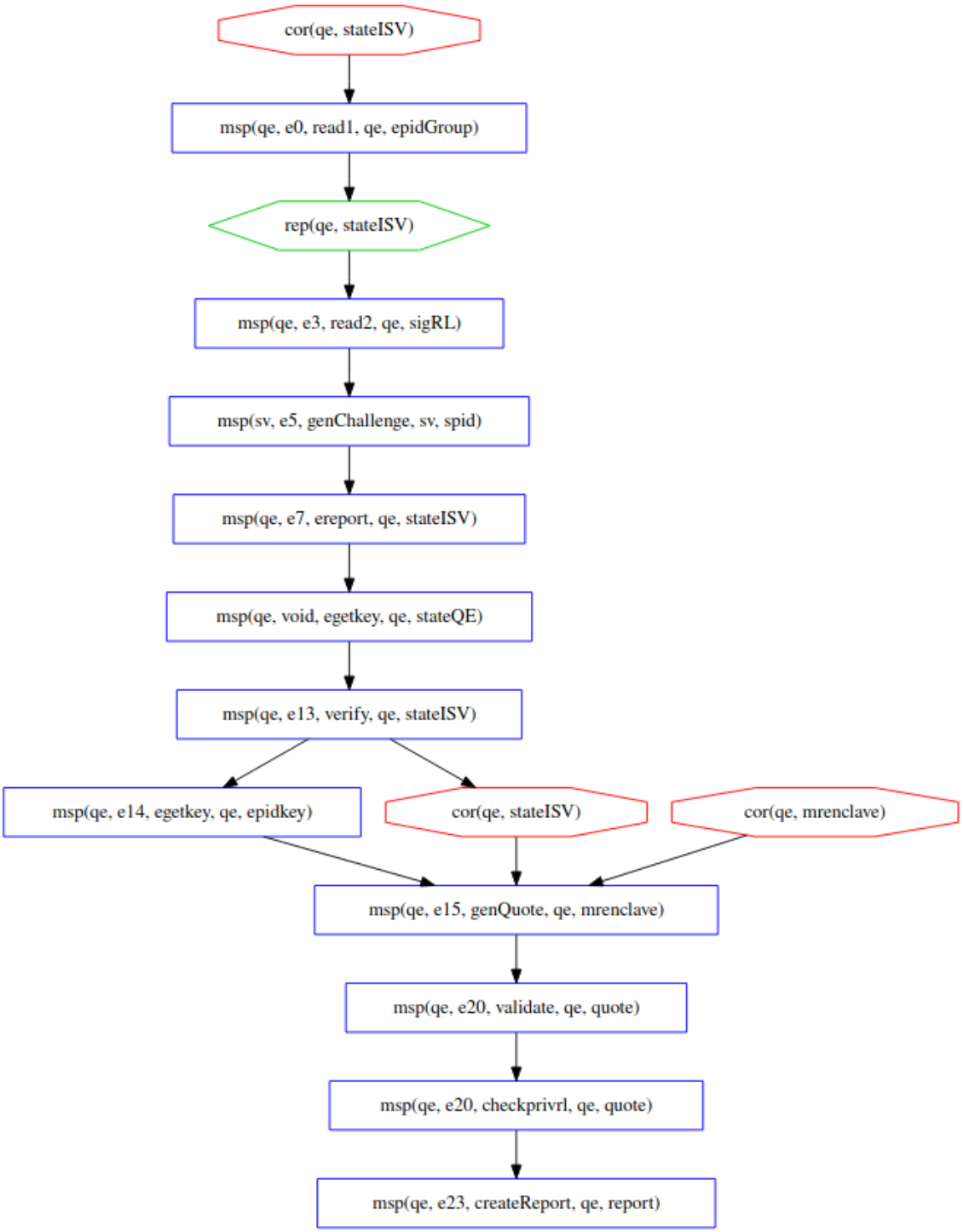

cor(qe, stateISV)
msp(qe, e0, read1, qe, epidGroup)
msp(ias, e3, read2, ias, sigRL)
msp(sv, e5, genChallenge, sv, spid)
rep(qe, stateISV)
msp(qe, e7, ereport, qe, stateISV)
msp(qe, void, egetkey, qe, stateQE)
msp(qe, e13, verify, qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, stateISV)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(ias, e20, validate, ias, quote)
msp(ias, e20, checkprivrl, ias, quote)
msp(ias, e23, createReport, ias, report)

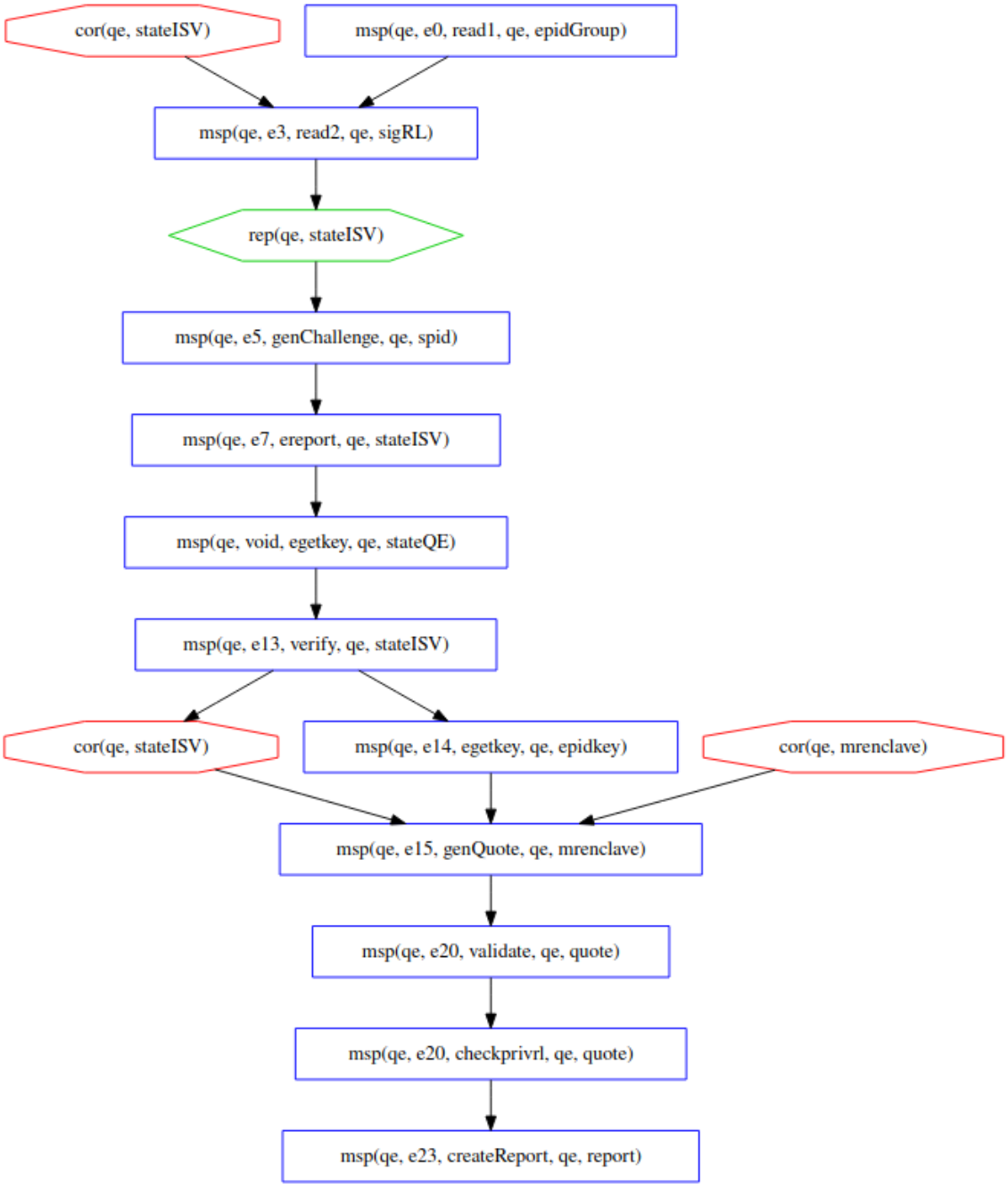

cor(qe, stateISV)
msp(qe, e0, read1, qe, epidGroup)
msp(ias, e3, read2, ias, sigRL)
msp(sv, e5, genChallenge, sv, spid)
rep(qe, stateISV)
msp(qe, e7, ereport, qe, stateISV)
msp(qe, void, egetkey, qe, stateQE)
msp(qe, e13, verify, qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, stateISV)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(ias, e20, validate, ias, quote)
msp(ias, e20, checkprivrl, ias, quote)
msp(ias, e23, createReport, ias, report)

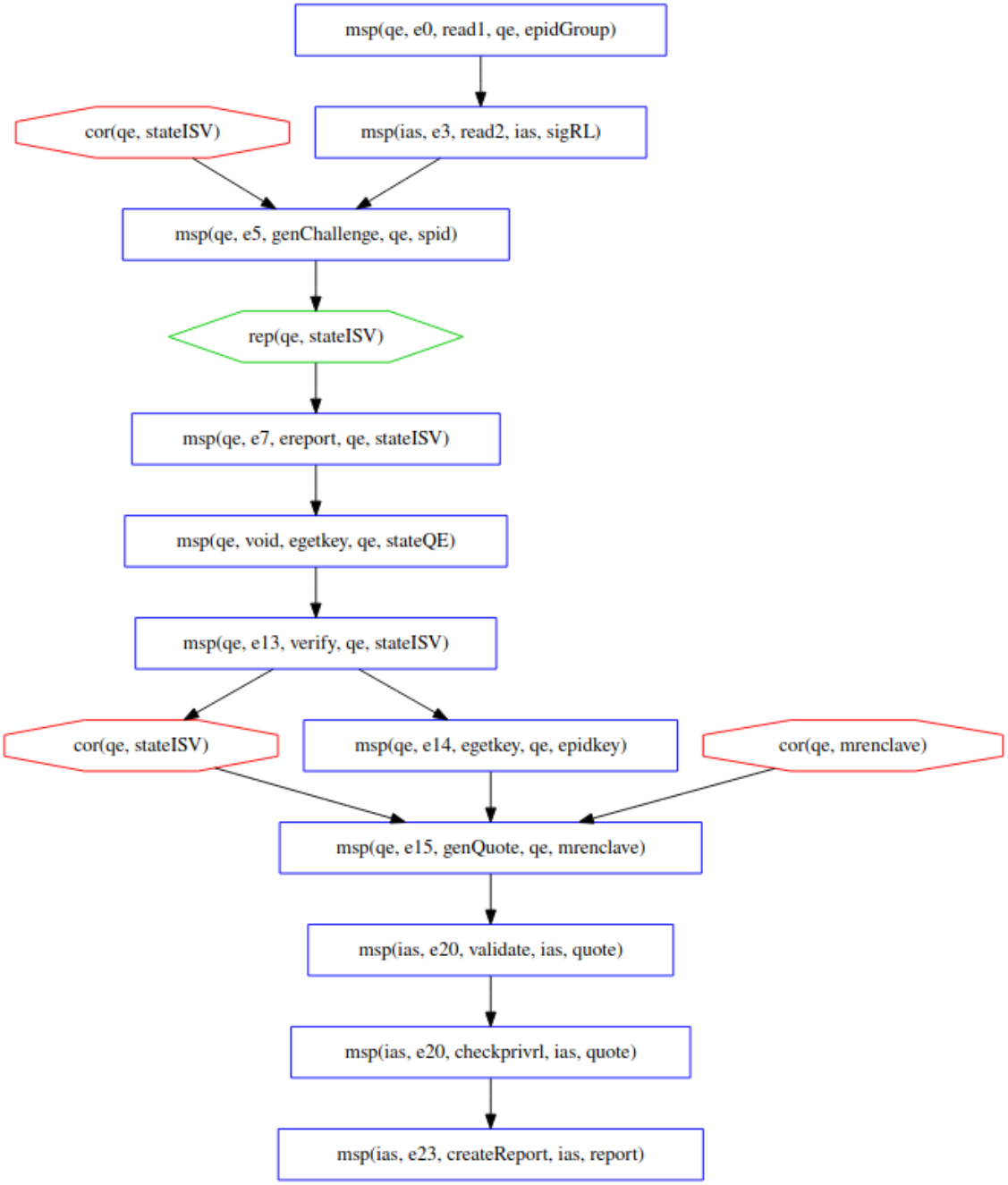

cor(qe, stateISV)
msp(qe, e0, read1, qe, epidGroup)
msp(ias, e3, read2, ias, sigRL)
rep(qe, stateISV)
msp(qe, e5, genChallenge, qe, spid)
msp(qe, e7, ereport, qe, stateISV)
msp(qe, void, egetkey, qe, stateQE)
msp(qe, e13, verify, qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, stateISV)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(ias, e20, validate, ias, quote)
msp(ias, e20, checkprivrl, ias, quote)
msp(ias, e23, createReport, ias, report)

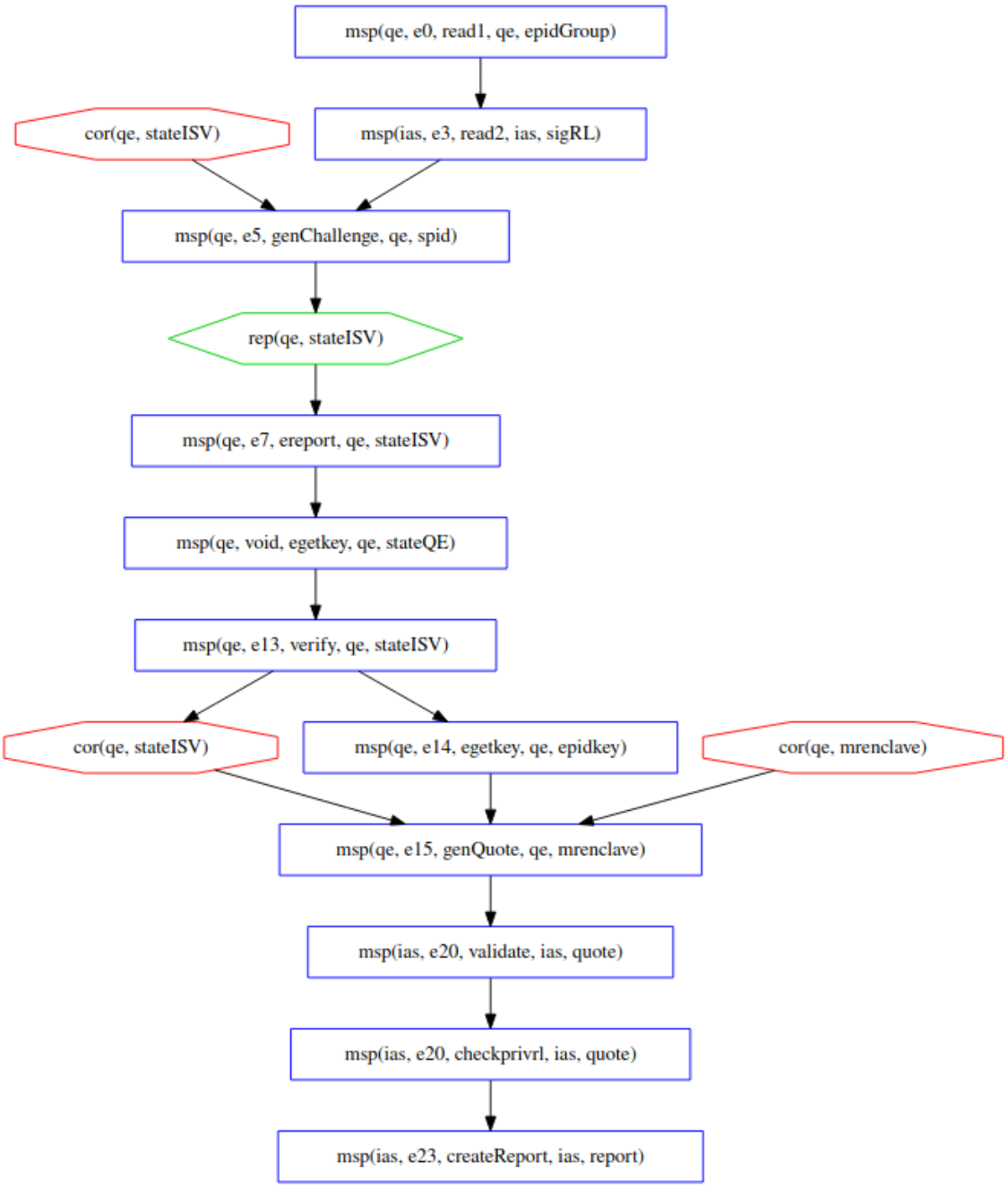

cor(qe, stateISV)
msp(qe, e0, read1, qe, epidGroup)
msp(qe, e3, read2, qe, sigRL)
msp(sv, e5, genChallenge, sv, spid)
rep(qe, stateISV)
msp(qe, e7, ereport, qe, stateISV)
msp(qe, void, egetkey, qe, stateQE)
msp(qe, e13, verify, qe, stateISV)
cor(qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(qe, e20, validate, qe, quote)
msp(qe, e20, checkprivrl, qe, quote)
msp(qe, e23, createReport, qe, report)

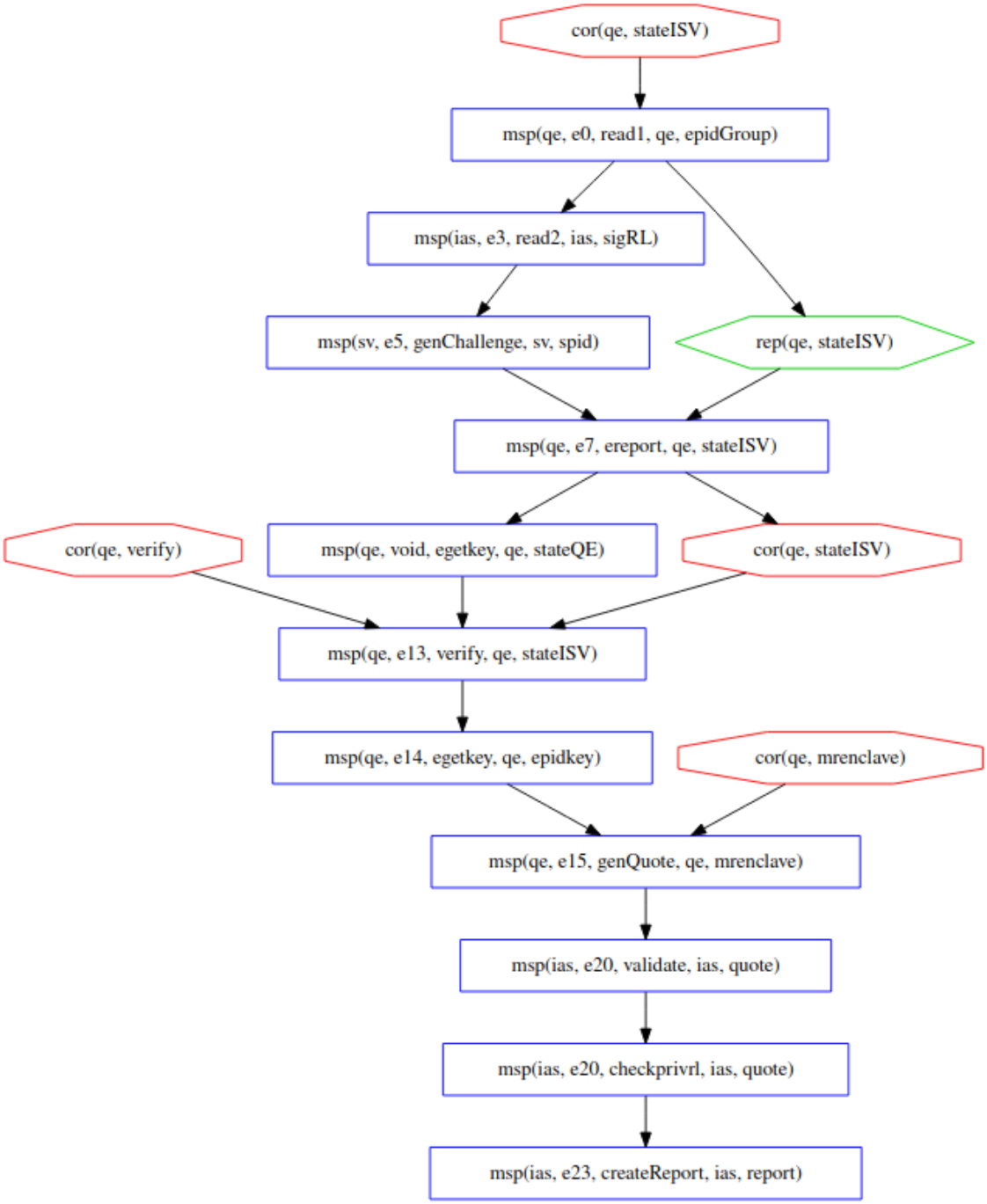

cor(qe, stateISV)
msp(qe, e0, read1, qe, epidGroup)
msp(qe, e3, read2, qe, sigRL)
msp(sv, e5, genChallenge, sv, spid)
rep(qe, stateISV)
msp(qe, e7, ereport, qe, stateISV)
msp(qe, void, egetkey, qe, stateQE)
msp(qe, e13, verify, qe, stateISV)
cor(qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(qe, e20, validate, qe, quote)
msp(qe, e20, checkprivrl, qe, quote)
msp(qe, e23, createReport, qe, report)

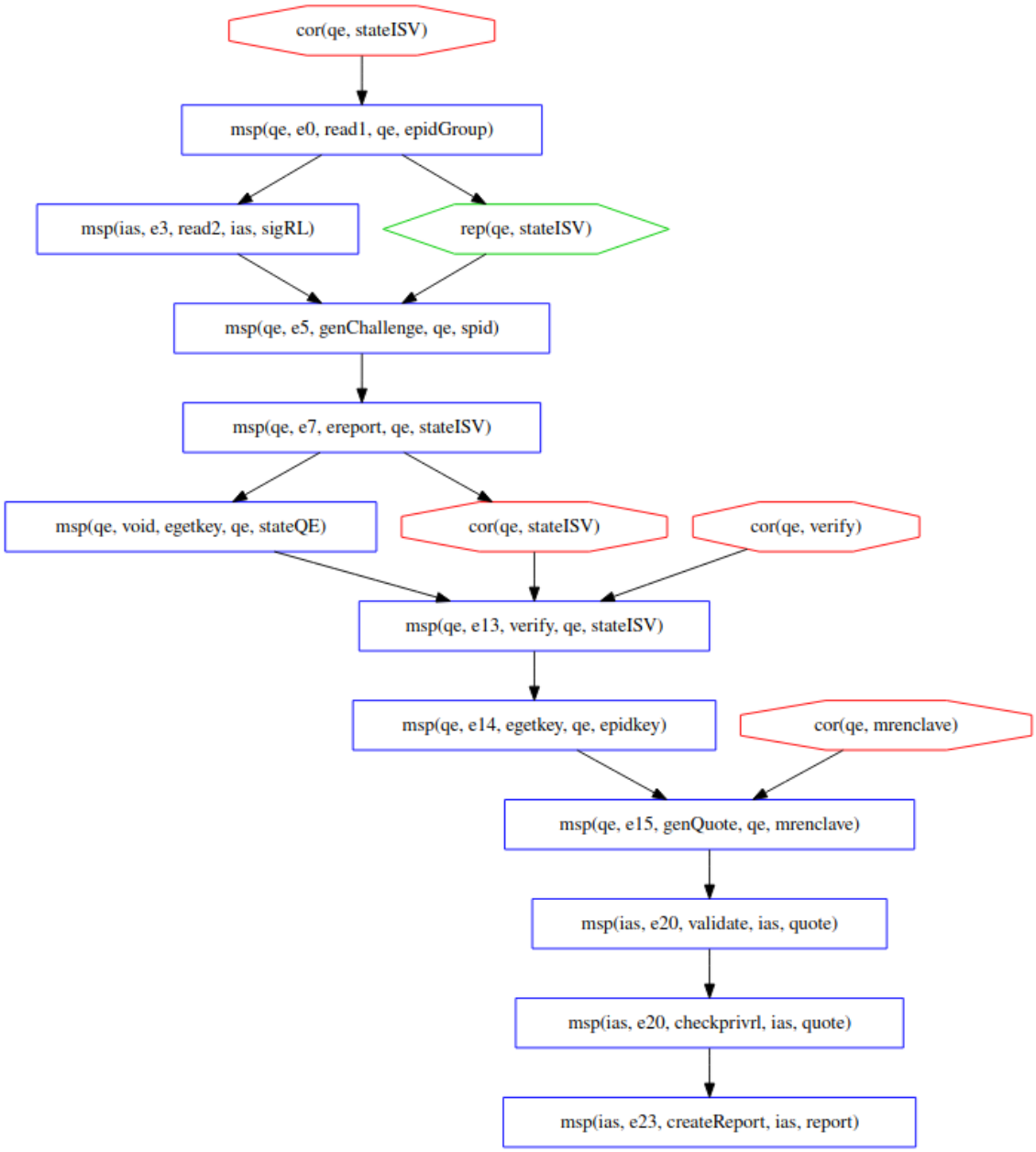

cor(qe, stateISV)
msp(qe, e0, read1, qe, epidGroup)
rep(qe, stateISV)
msp(qe, e3, read2, qe, sigRL)
msp(sv, e5, genChallenge, sv, spid)
msp(qe, e7, ereport, qe, stateISV)
msp(qe, void, egetkey, qe, stateQE)
msp(qe, e13, verify, qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, stateISV)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(qe, e20, validate, qe, quote)
msp(qe, e20, checkprivrl, qe, quote)
msp(qe, e23, createReport, qe, report)

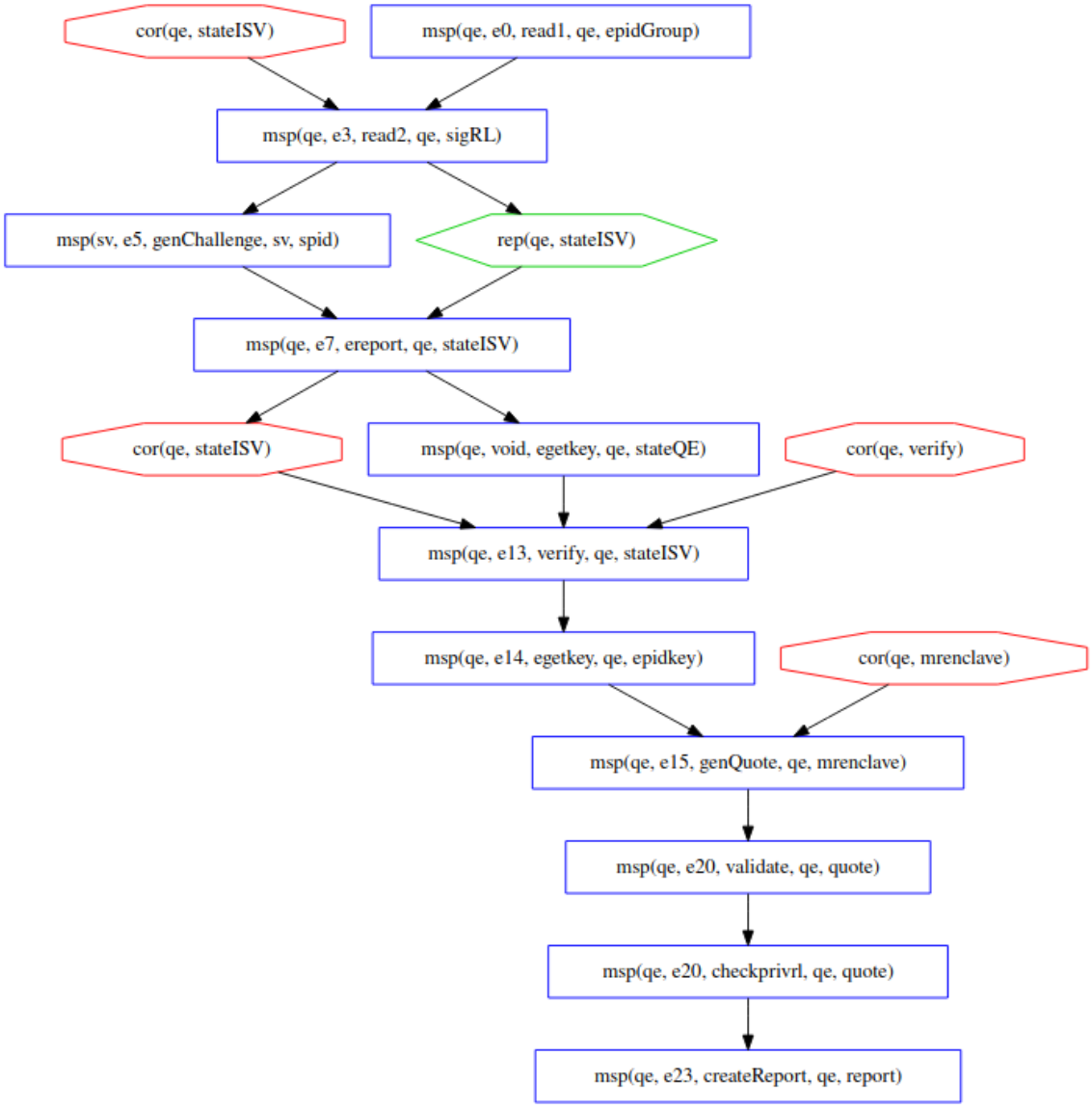

cor(qe, stateISV)
msp(qe, e0, read1, qe, epidGroup)
msp(qe, e3, read2, qe, sigRL)
rep(qe, stateISV)
msp(qe, e5, genChallenge, qe, spid)
msp(qe, e7, ereport, qe, stateISV)
msp(qe, void, egetkey, qe, stateQE)
msp(qe, e13, verify, qe, stateISV)
cor(qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(qe, e20, validate, qe, quote)
msp(qe, e20, checkprivrl, qe, quote)
msp(qe, e23, createReport, qe, report)

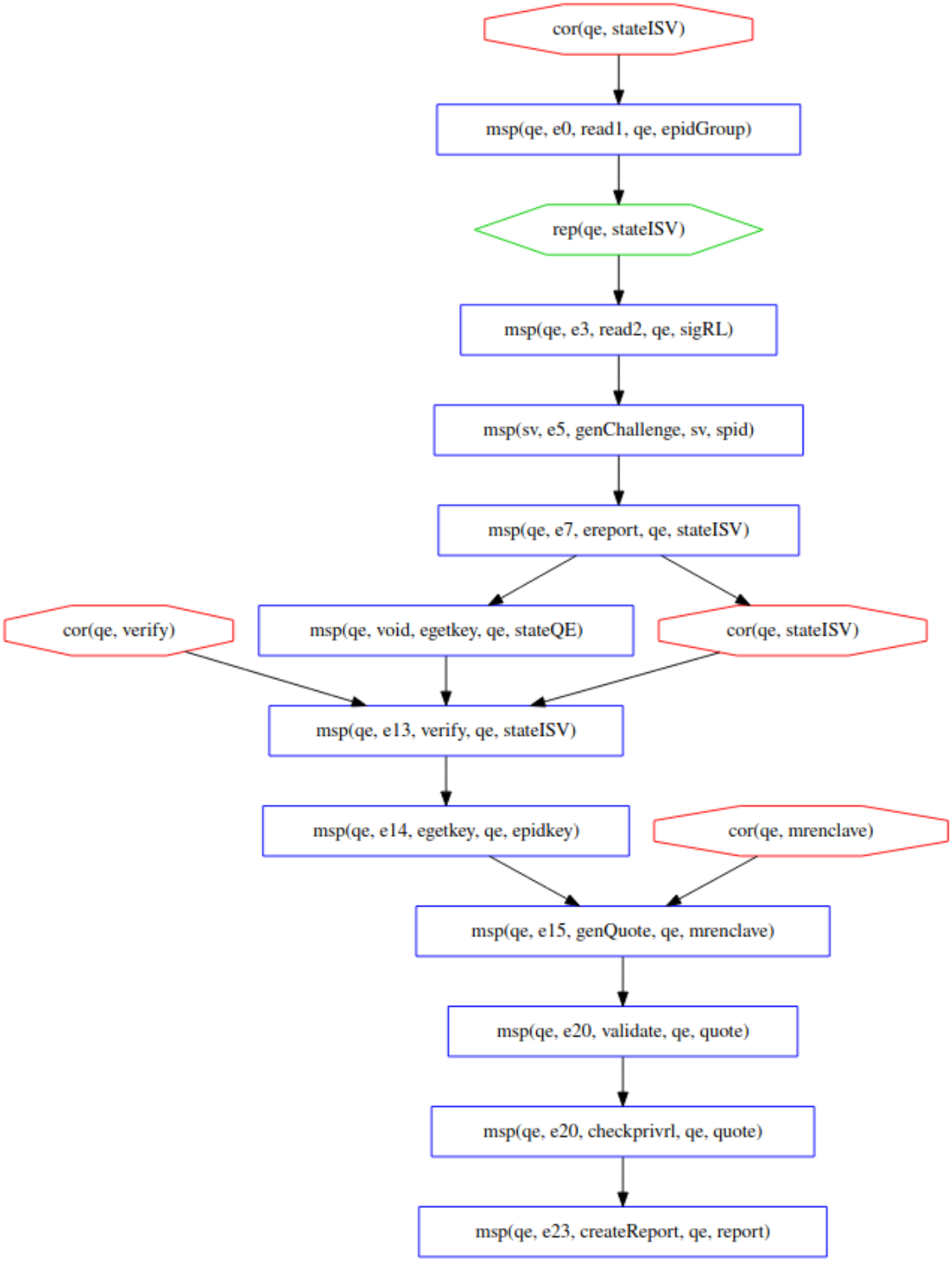

msp(qe, e0, read1, qe, epidGroup)
cor(qe, stateISV)
msp(ias, e3, read2, ias, sigRL)
msp(qe, e5, genChallenge, qe, spid)
rep(qe, stateISV)
msp(qe, e7, ereport, qe, stateISV)
msp(qe, void, egetkey, qe, stateQE)
msp(qe, e13, verify, qe, stateISV)
cor(qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(ias, e20, validate, ias, quote)
msp(ias, e20, checkprivrl, ias, quote)
msp(ias, e23, createReport, ias, report)

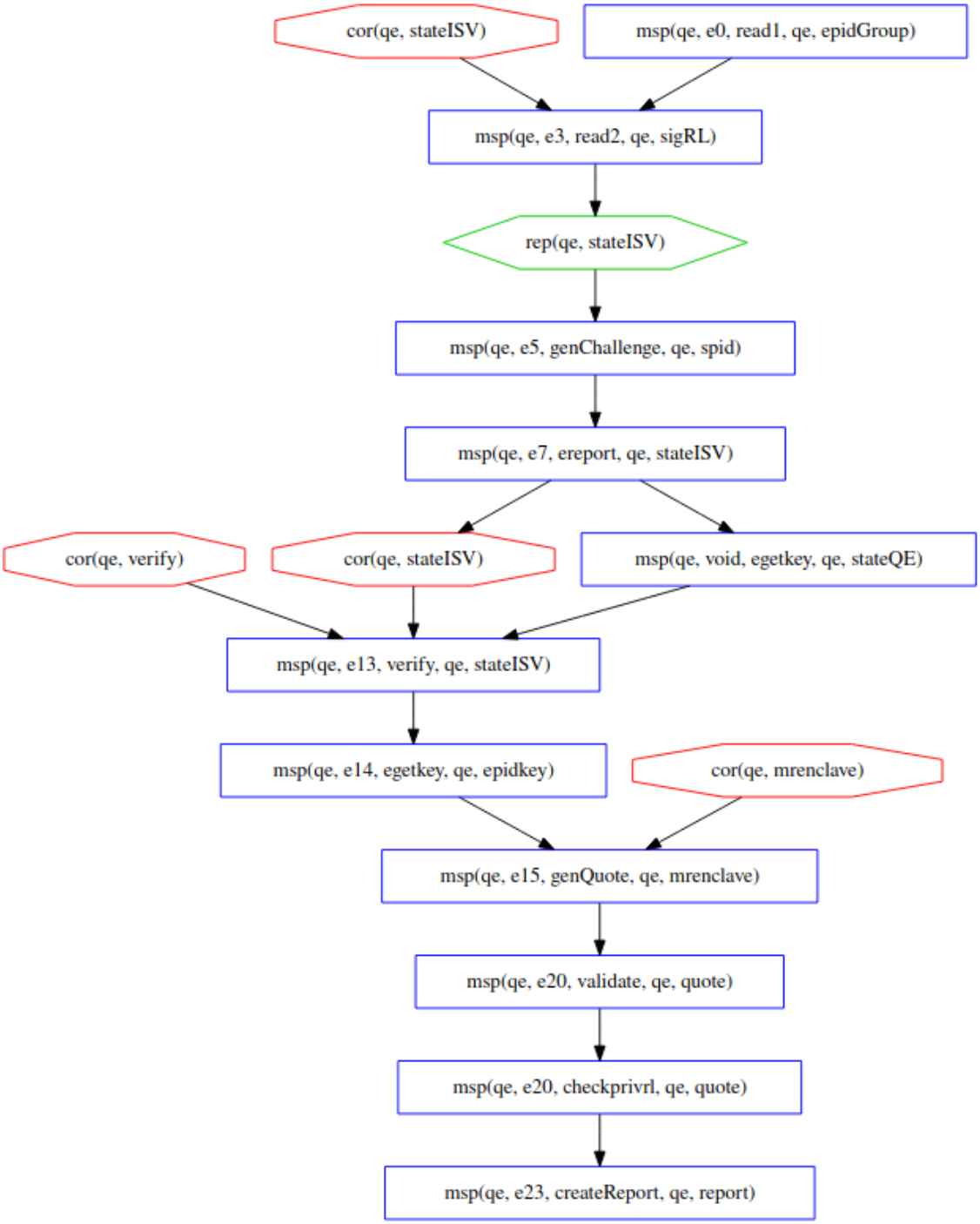

msp(qe, e0, read1, qe, epidGroup)
cor(qe, stateISV)
msp(ias, e3, read2, ias, sigRL)
msp(qe, e5, genChallenge, qe, spid)
rep(qe, stateISV)
msp(qe, e7, ereport, qe, stateISV)
msp(qe, void, egetkey, qe, stateQE)
msp(qe, e13, verify, qe, stateISV)
cor(qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(ias, e20, validate, ias, quote)
msp(ias, e20, checkprivrl, ias, quote)
msp(ias, e23, createReport, ias, report)

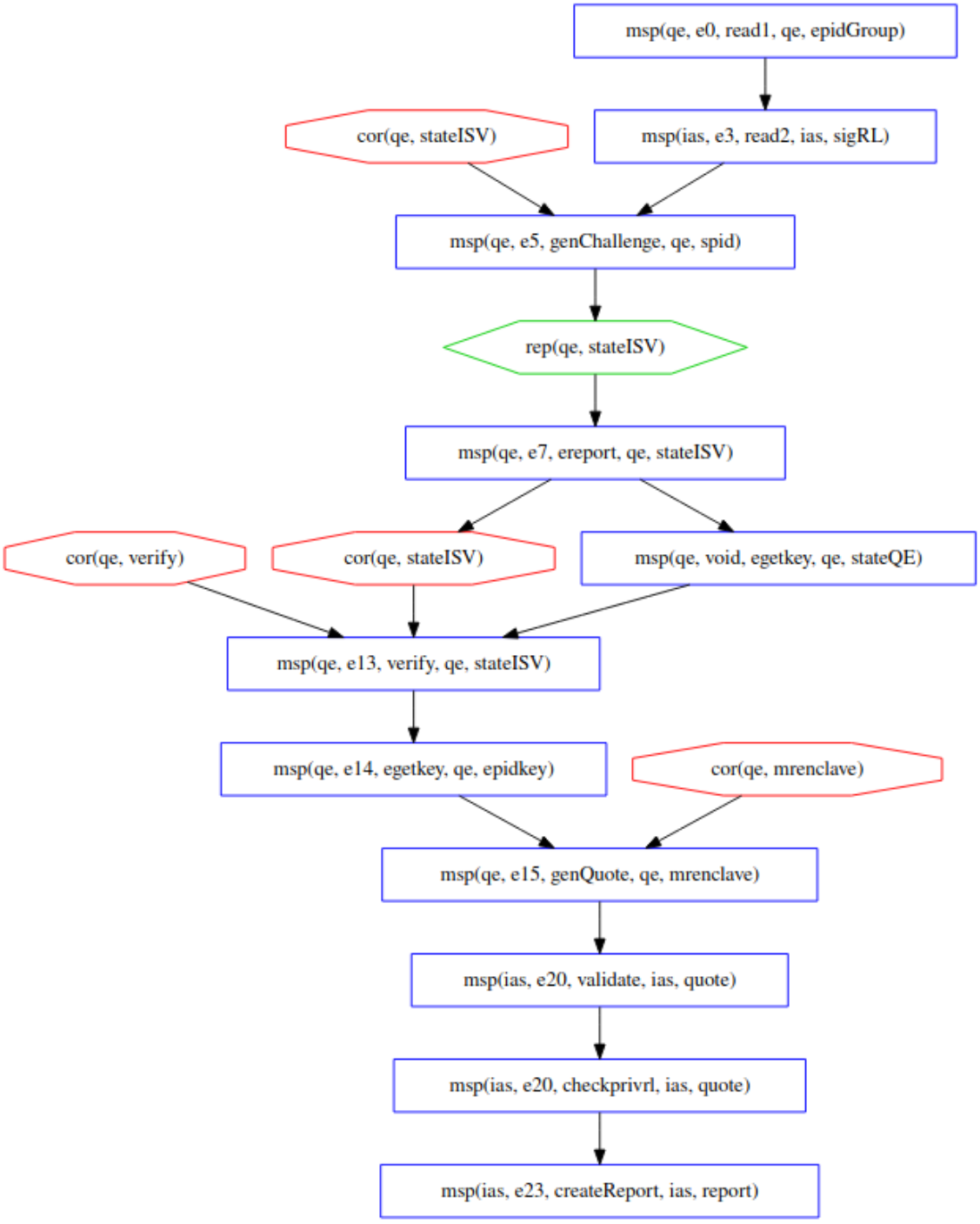

cor(qe, stateISV)
msp(qe, e0, read1, qe, epidGroup)
msp(ias, e3, read2, ias, sigRL)
msp(sv, e5, genChallenge, sv, spid)
rep(qe, stateISV)
msp(qe, e7, ereport, qe, stateISV)
cor(qe, verify)
msp(qe, void, egetkey, qe, stateQE)
cor(qe, stateISV)
msp(qe, e13, verify, qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(ias, e20, validate, ias, quote)
msp(ias, e20, checkprivrl, ias, quote)
msp(ias, e23, createReport, ias, report)

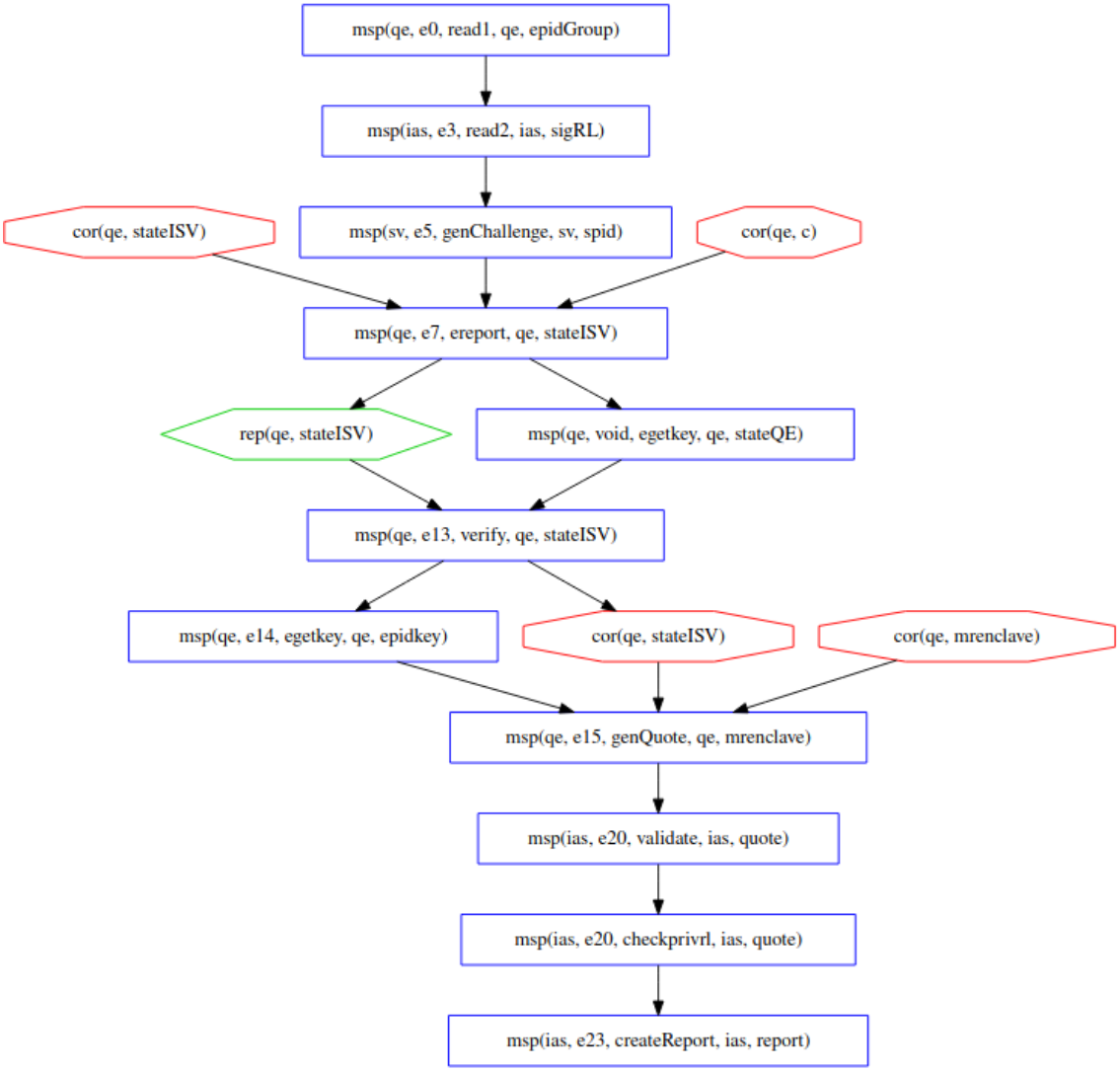

cor(qe, stateISV)
msp(qe, e0, read1, qe, epidGroup)
msp(ias, e3, read2, ias, sigRL)
rep(qe, stateISV)
msp(qe, e5, genChallenge, qe, spid)
msp(qe, e7, ereport, qe, stateISV)
msp(qe, void, egetkey, qe, stateQE)
cor(qe, stateISV)
cor(qe, verify)
msp(qe, e13, verify, qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(ias, e20, validate, ias, quote)
msp(ias, e20, checkprivrl, ias, quote)
msp(ias, e23, createReport, ias, report)

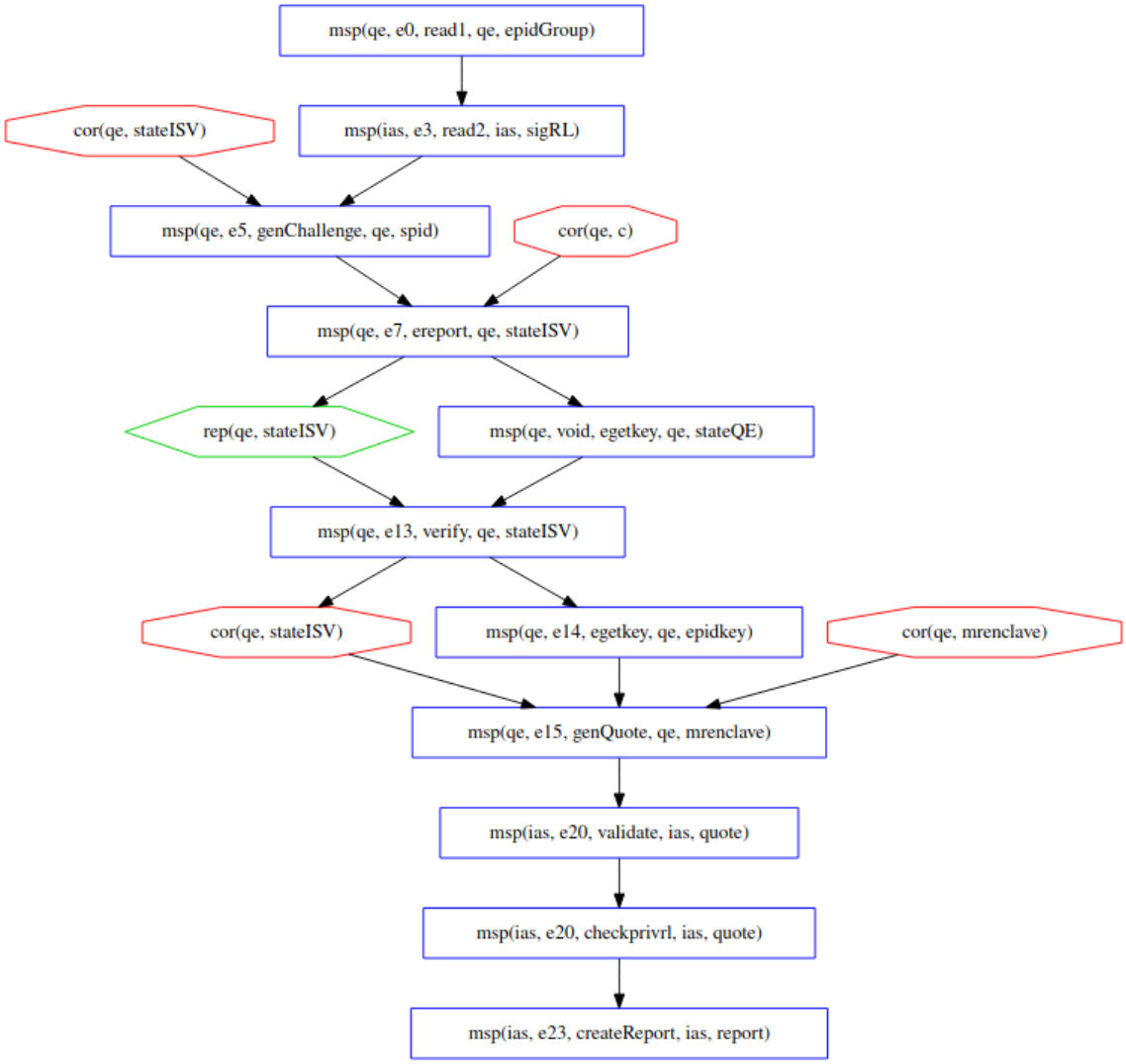

cor(qe, stateISV)
msp(qe, e0, read1, qe, epidGroup)
msp(qe, e3, read2, qe, sigRL)
msp(sv, e5, genChallenge, sv, spid)
rep(qe, stateISV)
msp(qe, e7, ereport, qe, stateISV)
cor(qe, stateISV)
msp(qe, void, egetkey, qe, stateQE)
cor(qe, verify)
msp(qe, e13, verify, qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(qe, e20, validate, qe, quote)
msp(qe, e20, checkprivrl, qe, quote)
msp(qe, e23, createReport, qe, report)

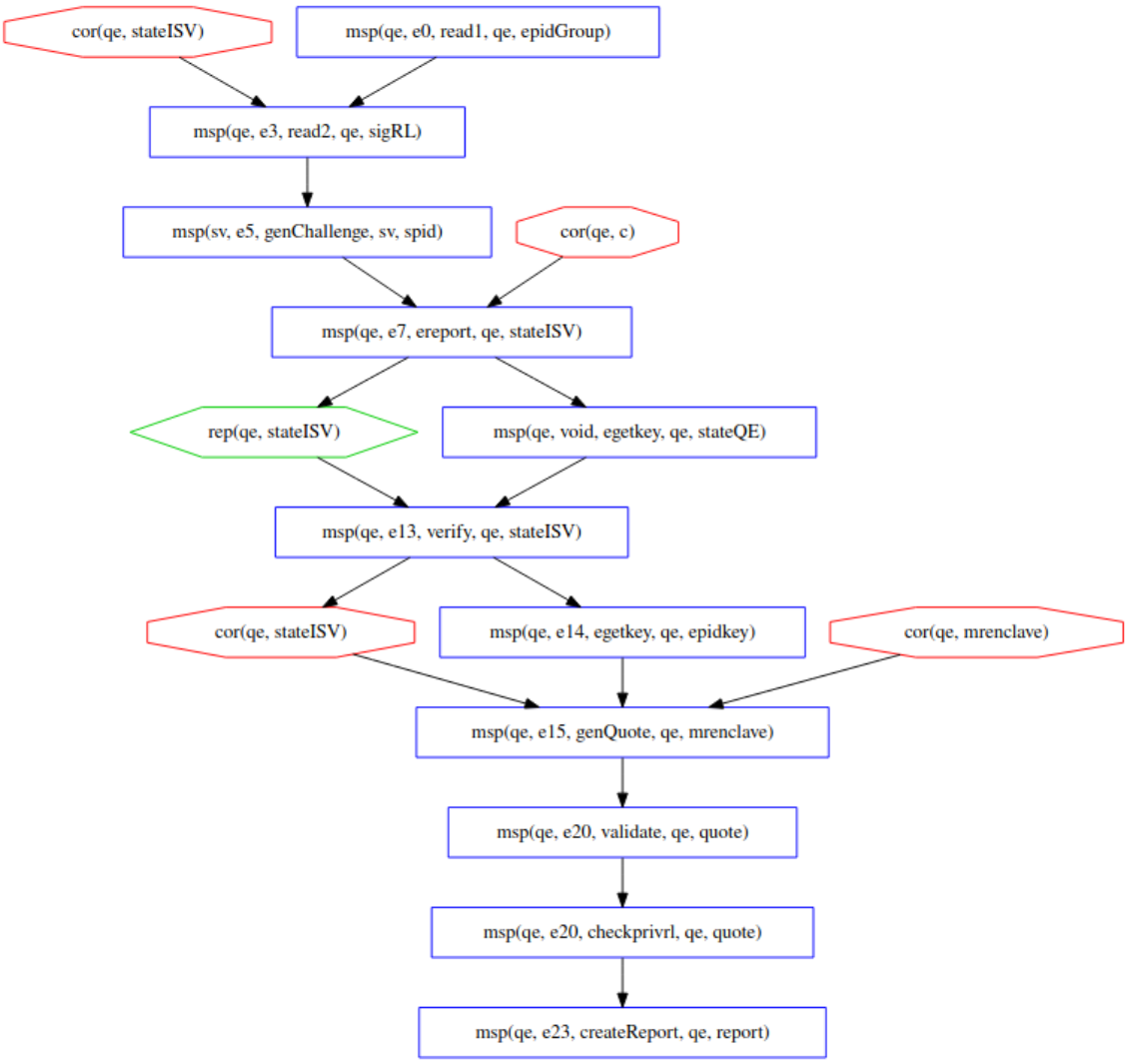

cor(qe, stateISV)
msp(qe, e0, read1, qe, epidGroup)
rep(qe, stateISV)
msp(qe, e3, read2, qe, sigRL)
msp(sv, e5, genChallenge, sv, spid)
msp(qe, e7, ereport, qe, stateISV)
cor(qe, verify)
msp(qe, void, egetkey, qe, stateQE)
cor(qe, stateISV)
msp(qe, e13, verify, qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(qe, e20, validate, qe, quote)
msp(qe, e20, checkprivrl, qe, quote)
msp(qe, e23, createReport, qe, report)

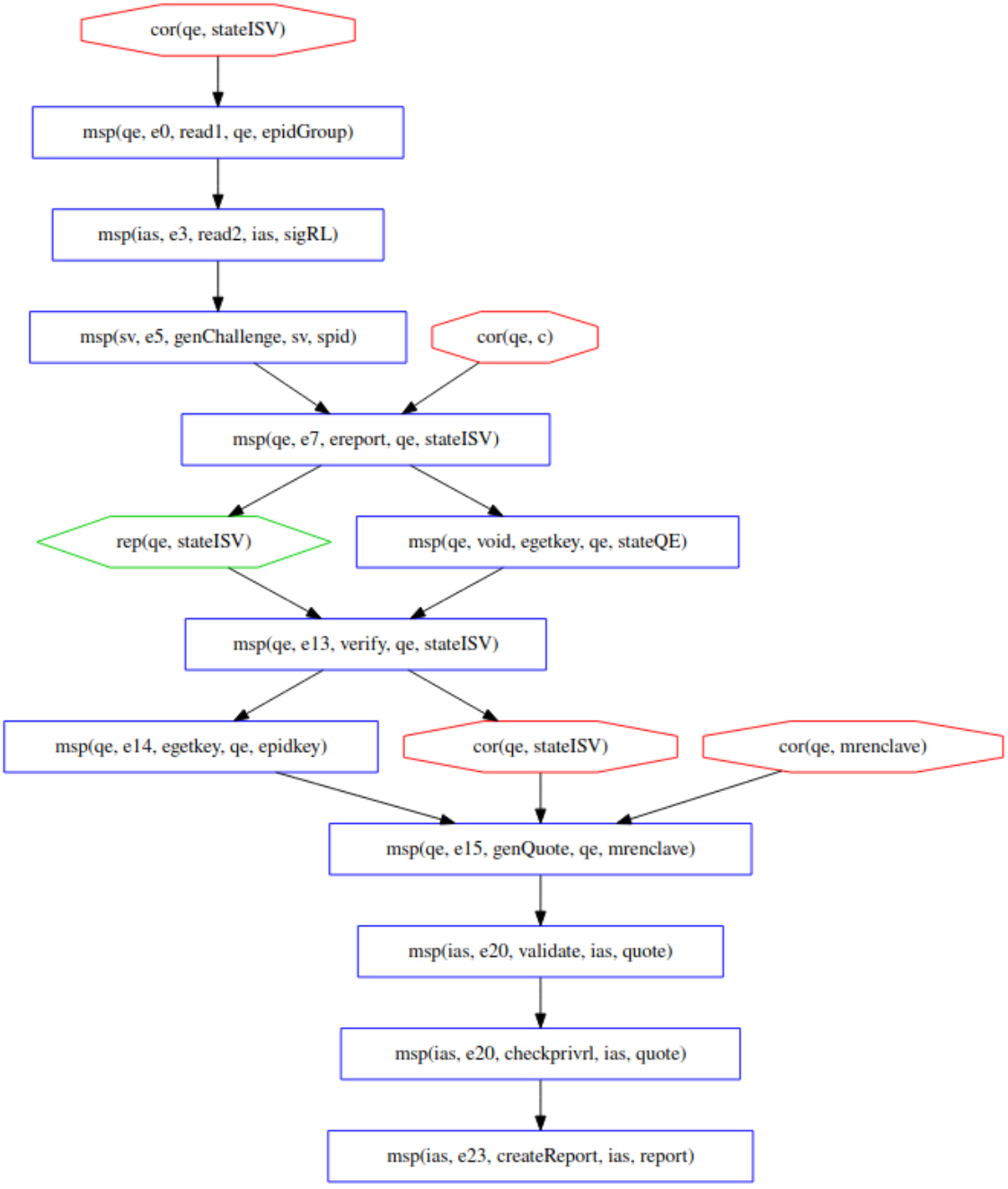

cor(qe, stateISV)
msp(qe, e0, read1, qe, epidGroup)
msp(qe, e3, read2, qe, sigRL)
rep(qe, stateISV)
msp(qe, e5, genChallenge, qe, spid)
msp(qe, e7, ereport, qe, stateISV)
cor(qe, verify)
cor(qe, stateISV)
msp(qe, void, egetkey, qe, stateQE)
msp(qe, e13, verify, qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(qe, e20, validate, qe, quote)
msp(qe, e20, checkprivrl, qe, quote)
msp(qe, e23, createReport, qe, report)

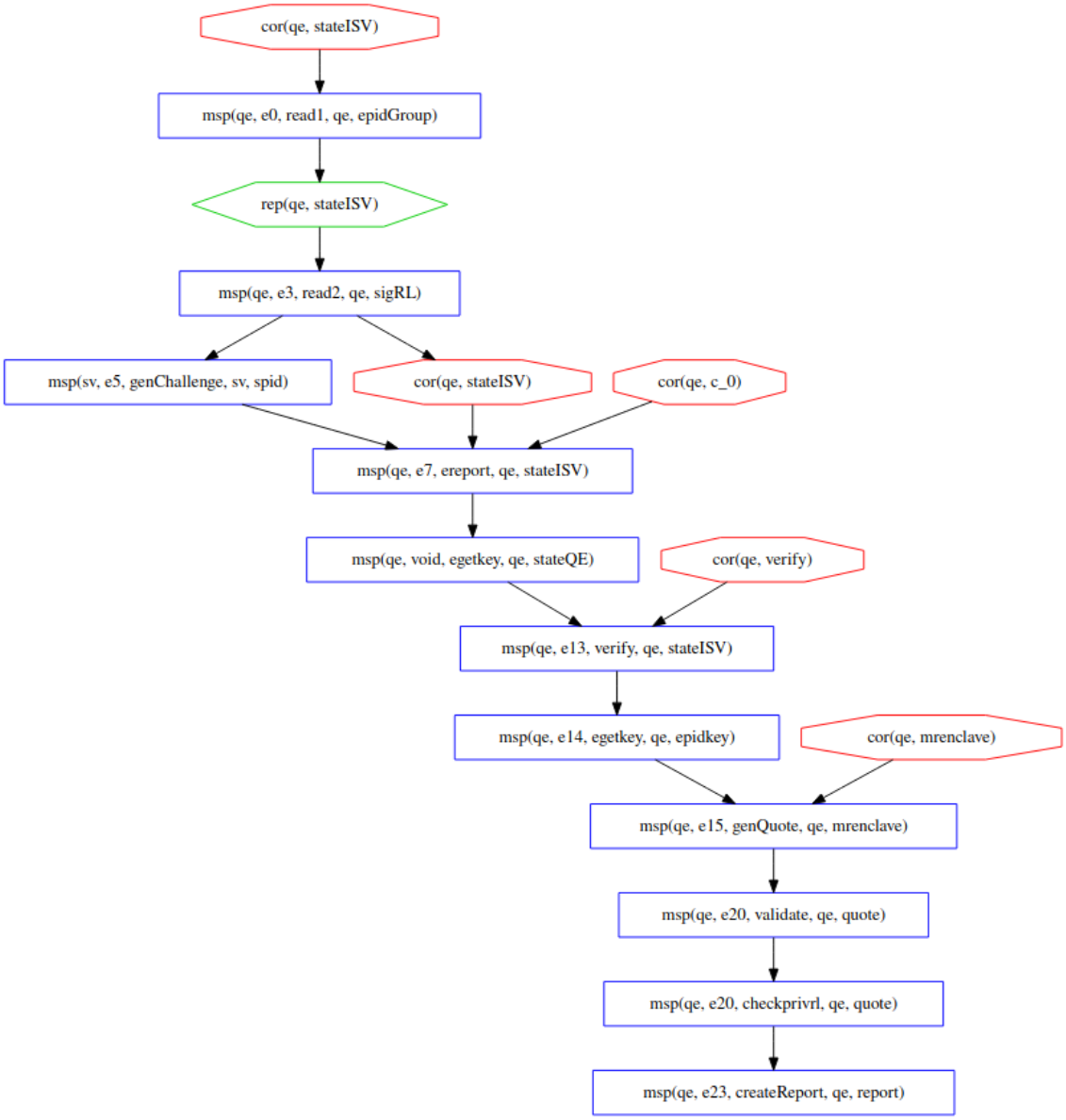

msp(qe, e0, read1, qe, epidGroup)
cor(qe, stateISV)
msp(ias, e3, read2, ias, sigRL)
msp(qe, e5, genChallenge, qe, spid)
rep(qe, stateISV)
msp(qe, e7, ereport, qe, stateISV)
cor(qe, verify)
cor(qe, stateISV)
msp(qe, void, egetkey, qe, stateQE)
msp(qe, e13, verify, qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(ias, e20, validate, ias, quote)
msp(ias, e20, checkprivrl, ias, quote)
msp(ias, e23, createReport, ias, report)

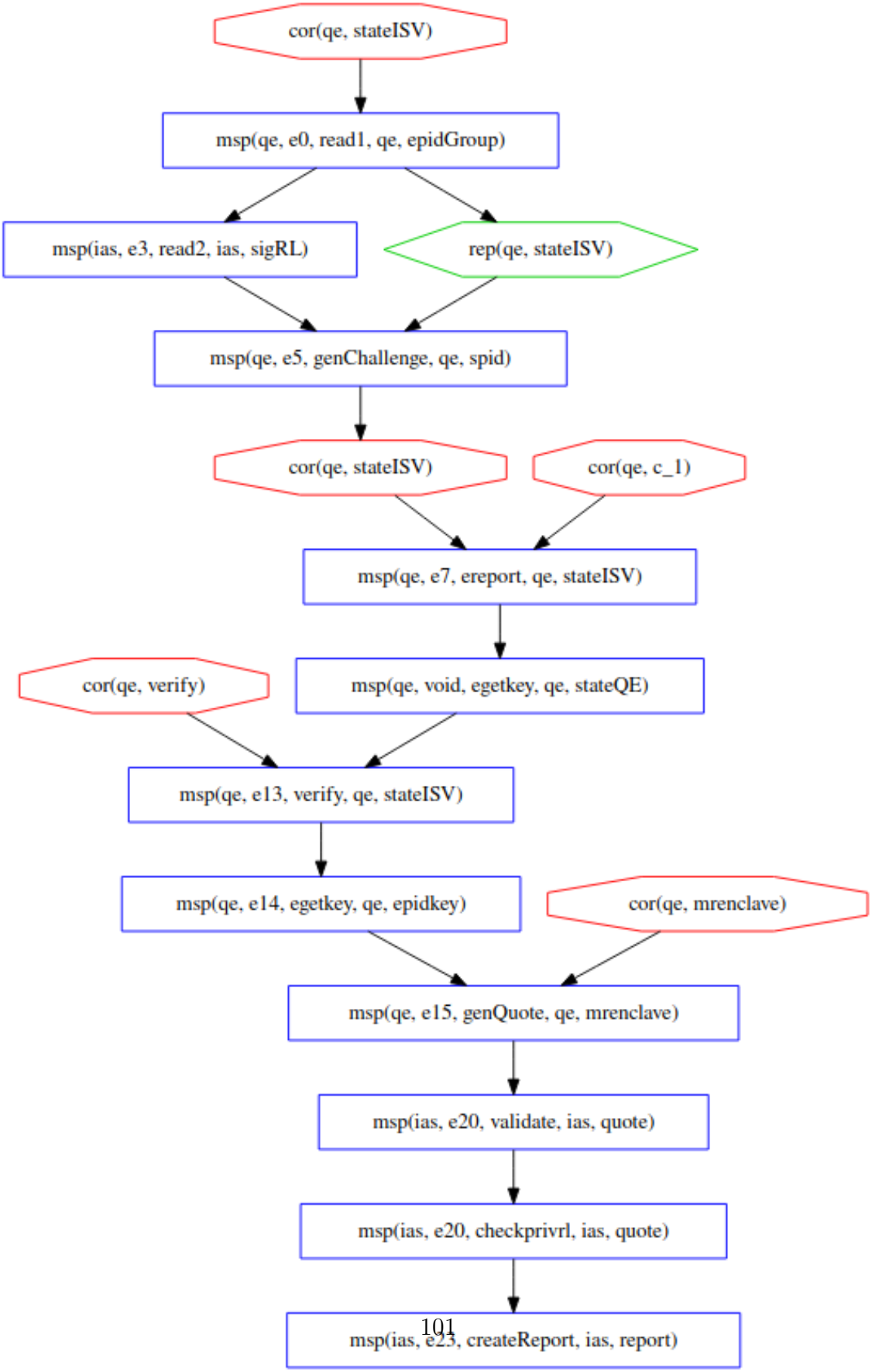

msp(qe, e0, read1, qe, epidGroup)
msp(ias, e3, read2, ias, sigRL)
cor(qe, stateISV)
msp(sv, e5, genChallenge, sv, spid)
cor(qe, c)
msp(qe, e7, ereport, qe, stateISV)
rep(qe, stateISV)
msp(qe, void, egetkey, qe, stateQE)
msp(qe, e13, verify, qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, stateISV)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(ias, e20, validate, ias, quote)
msp(ias, e20, checkprivrl, ias, quote)
msp(ias, e23, createReport, ias, report)

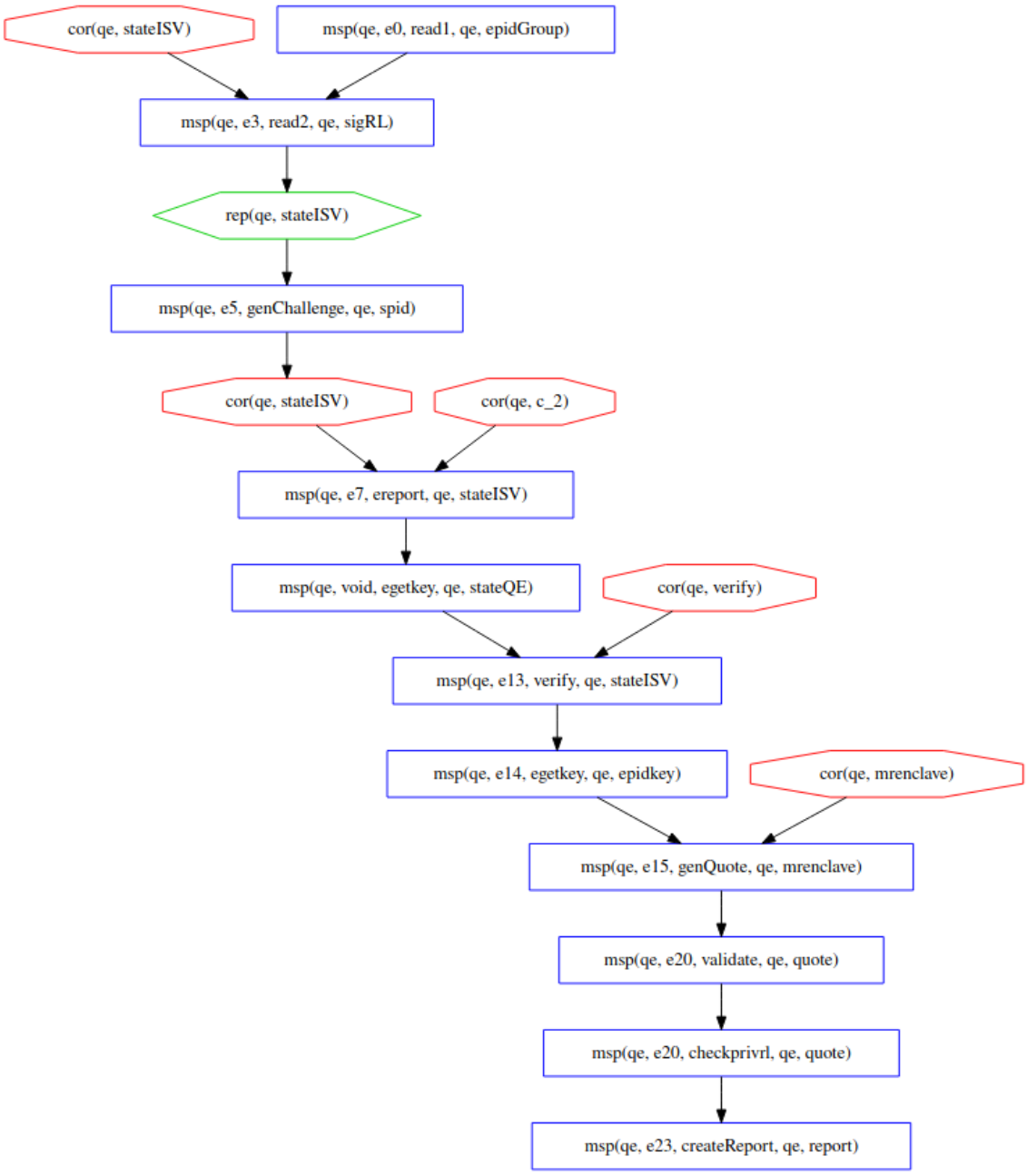

msp(qe, e0, read1, qe, epidGroup)
cor(qe, stateISV)
msp(ias, e3, read2, ias, sigRL)
msp(qe, e5, genChallenge, qe, spid)
cor(qe, c)
msp(qe, e7, ereport, qe, stateISV)
rep(qe, stateISV)
msp(qe, void, egetkey, qe, stateQE)
msp(qe, e13, verify, qe, stateISV)
cor(qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(ias, e20, validate, ias, quote)
msp(ias, e20, checkprivrl, ias, quote)
msp(ias, e23, createReport, ias, report)

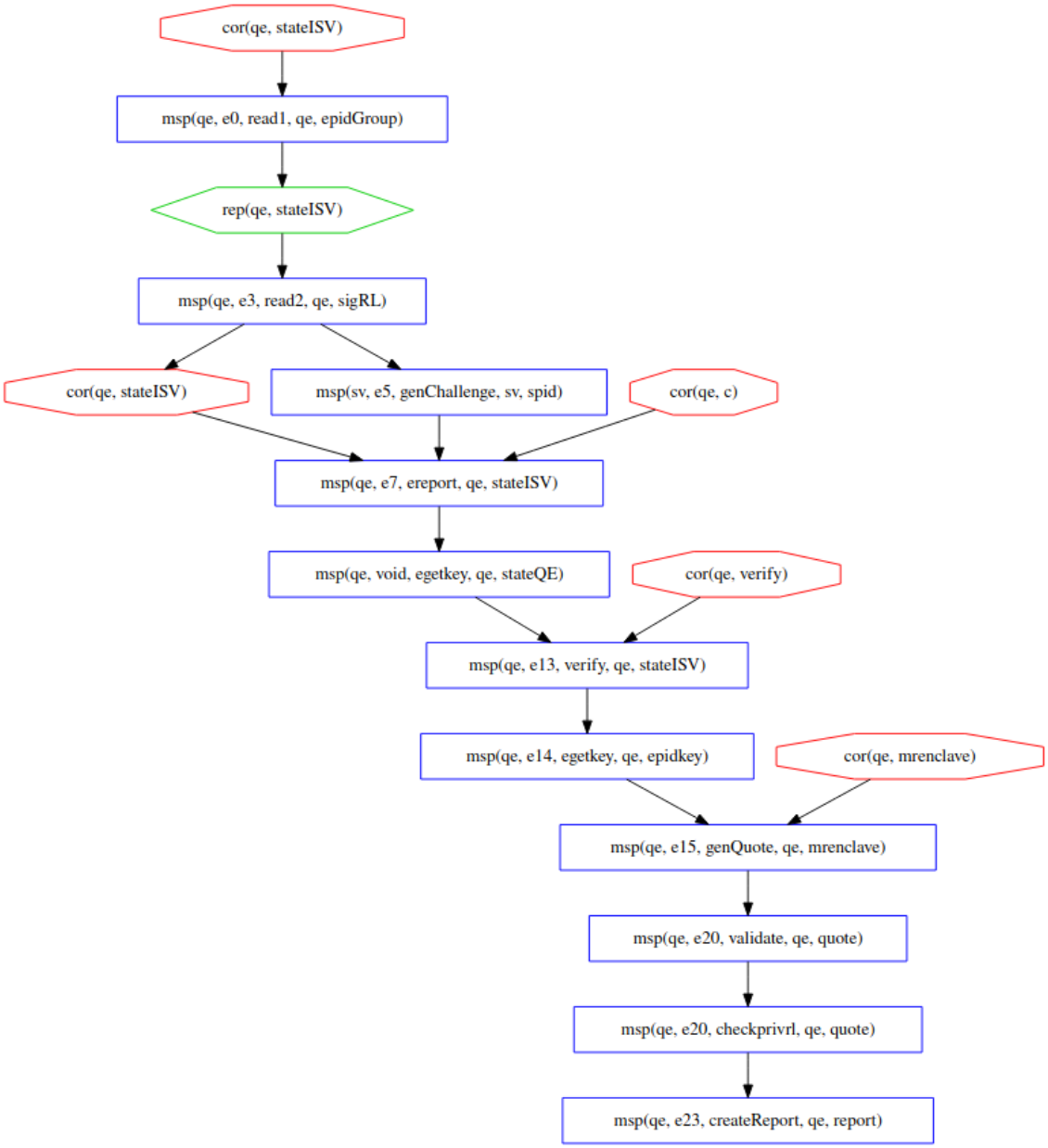

cor(qe, stateISV)
msp(qe, e0, read1, qe, epidGroup)
msp(qe, e3, read2, qe, sigRL)
msp(sv, e5, genChallenge, sv, spid)
cor(qe, c)
msp(qe, e7, ereport, qe, stateISV)
rep(qe, stateISV)
msp(qe, void, egetkey, qe, stateQE)
msp(qe, e13, verify, qe, stateISV)
cor(qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(qe, e20, validate, qe, quote)
msp(qe, e20, checkprivrl, qe, quote)
msp(qe, e23, createReport, qe, report)

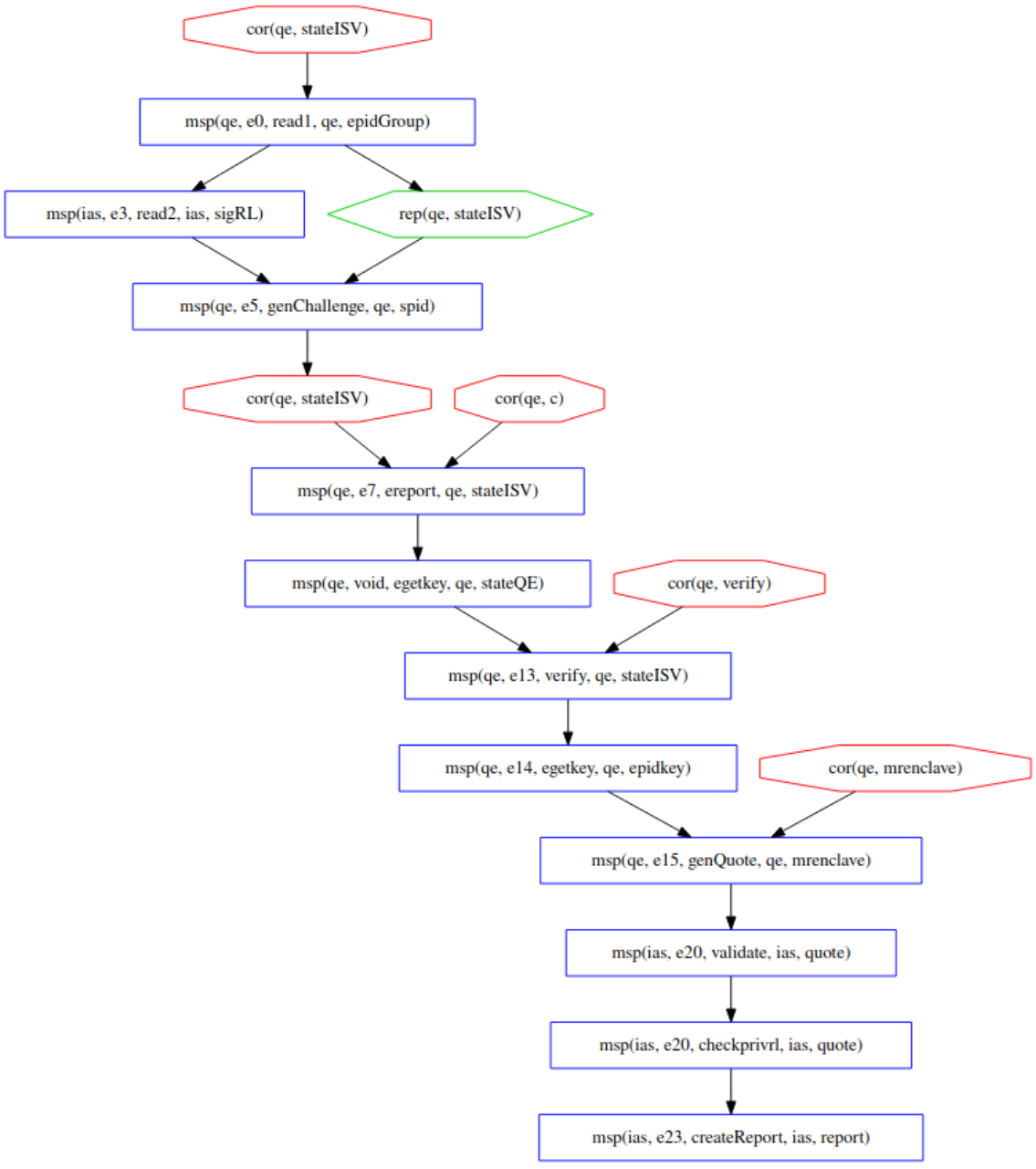

cor(qe, stateISV)
msp(qe, e0, read1, qe, epidGroup)
msp(ias, e3, read2, ias, sigRL)
msp(sv, e5, genChallenge, sv, spid)
cor(qe, c)
msp(qe, e7, ereport, qe, stateISV)
rep(qe, stateISV)
msp(qe, void, egetkey, qe, stateQE)
msp(qe, e13, verify, qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, stateISV)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(ias, e20, validate, ias, quote)
msp(ias, e20, checkprivrl, ias, quote)
msp(ias, e23, createReport, ias, report)

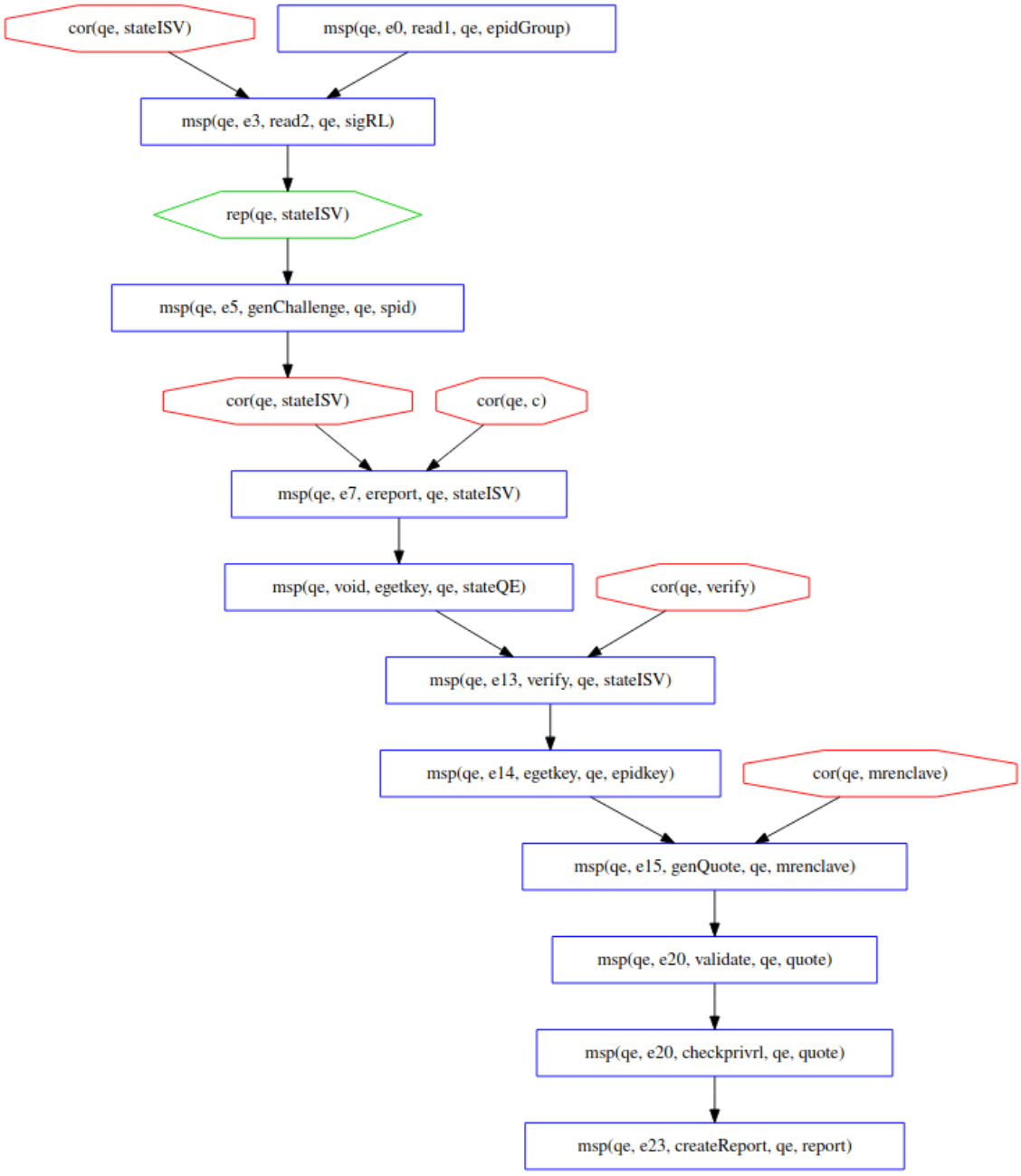

cor(qe, stateISV)
msp(qe, e0, read1, qe, epidGroup)
rep(qe, stateISV)
msp(qe, e3, read2, qe, sigRL)
msp(sv, e5, genChallenge, sv, spid)
cor(qe, stateISV)
cor(qe, c_0)
msp(qe, e7, ereport, qe, stateISV)
msp(qe, void, egetkey, qe, stateQE)
cor(qe, verify)
msp(qe, e13, verify, qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(qe, e20, validate, qe, quote)
msp(qe, e20, checkprivrl, qe, quote)
msp(qe, e23, createReport, qe, report)

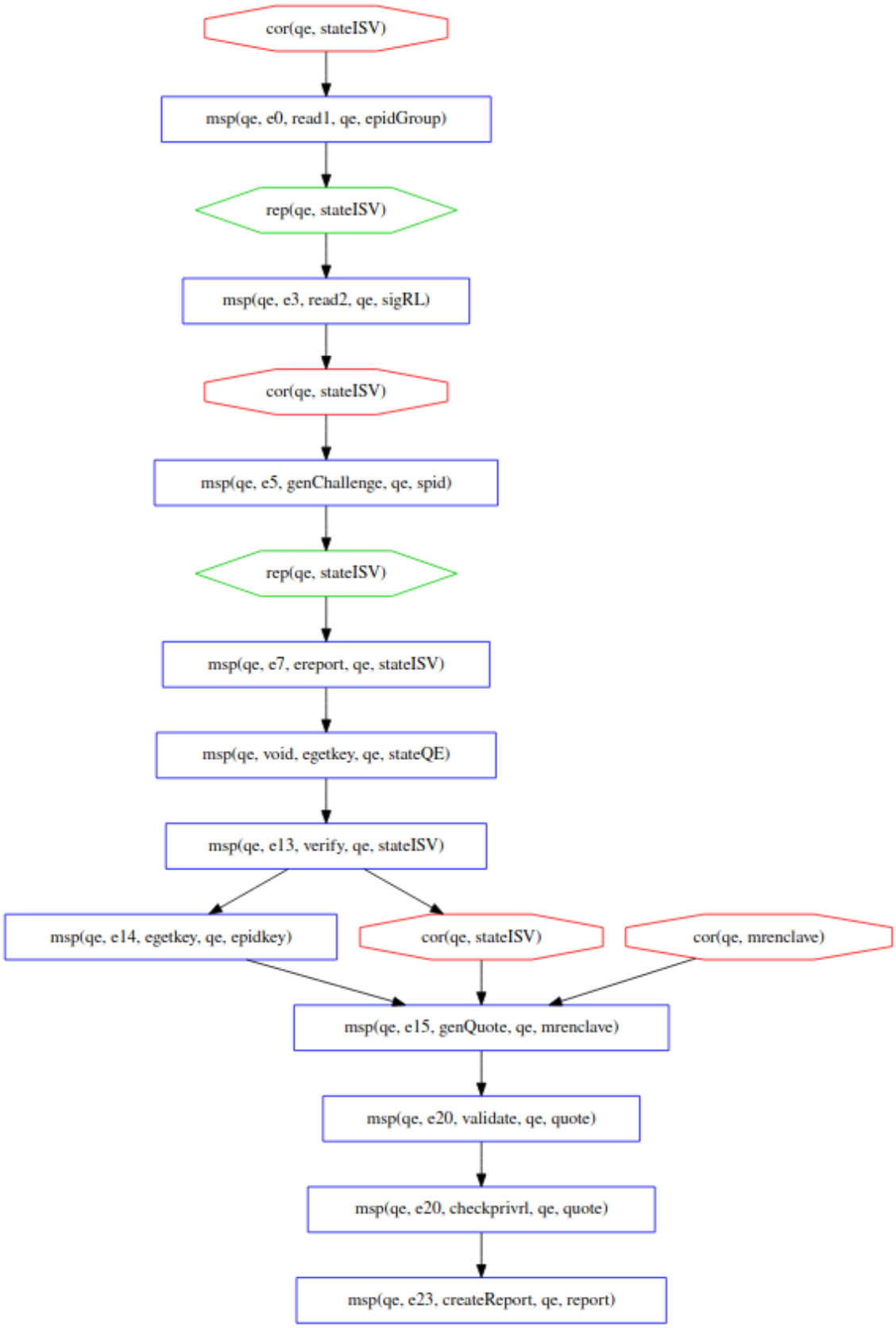

cor(qe, stateISV)
msp(qe, e0, read1, qe, epidGroup)
msp(ias, e3, read2, ias, sigRL)
rep(qe, stateISV)
msp(qe, e5, genChallenge, qe, spid)
cor(qe, stateISV)
cor(qe, c_1)
msp(qe, e7, ereport, qe, stateISV)
cor(qe, verify)
msp(qe, void, egetkey, qe, stateQE)
msp(qe, e13, verify, qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(ias, e20, validate, ias, quote)
msp(ias, e20, checkprivrl, ias, quote)
msp(ias, e23, createReport, ias, report)

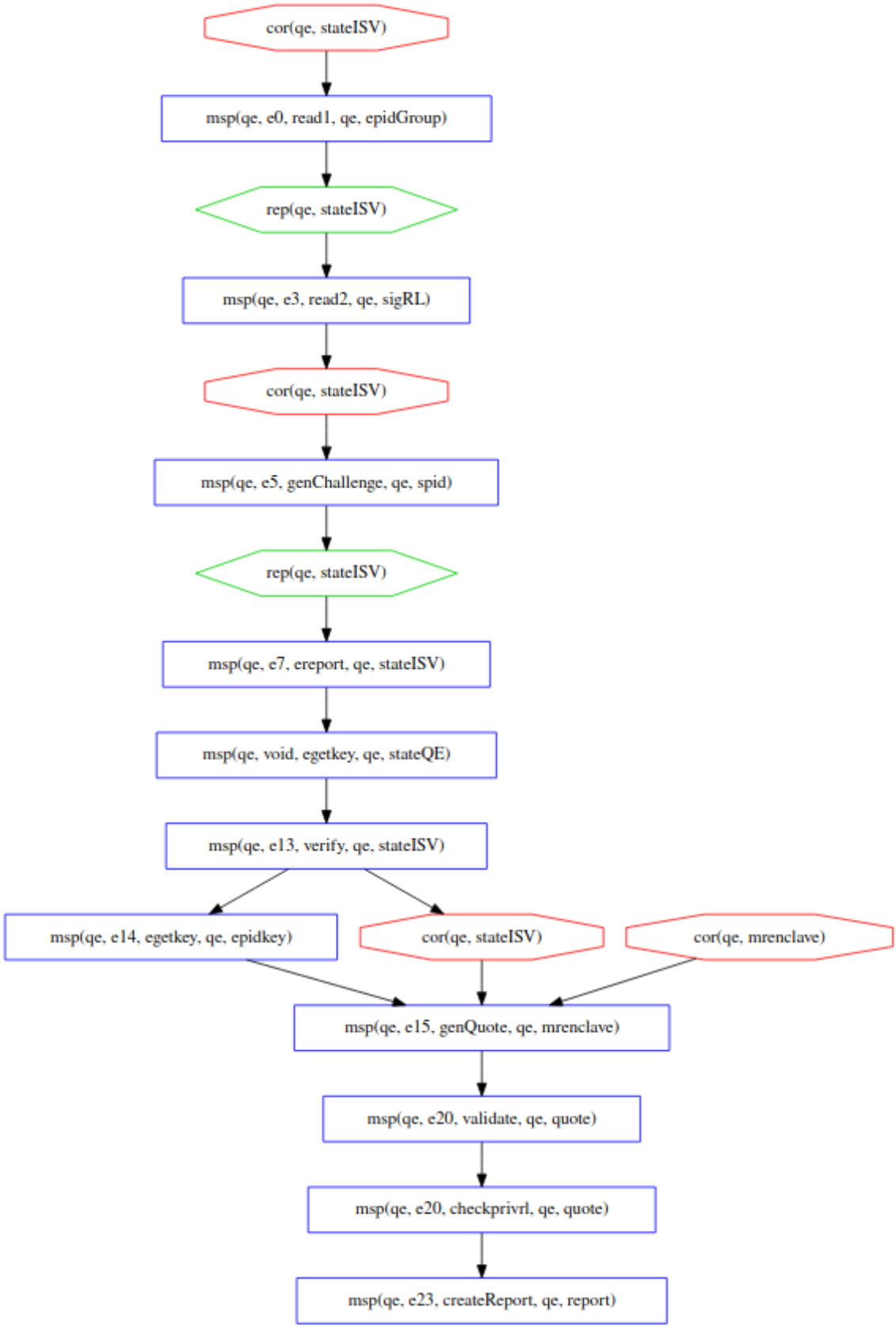

cor(qe, stateISV)
msp(qe, e0, read1, qe, epidGroup)
msp(qe, e3, read2, qe, sigRL)
rep(qe, stateISV)
msp(qe, e5, genChallenge, qe, spid)
cor(qe, stateISV)
cor(qe, c_2)
msp(qe, e7, ereport, qe, stateISV)
msp(qe, void, egetkey, qe, stateQE)
cor(qe, verify)
msp(qe, e13, verify, qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(qe, e20, validate, qe, quote)
msp(qe, e20, checkprivrl, qe, quote)
msp(qe, e23, createReport, qe, report)

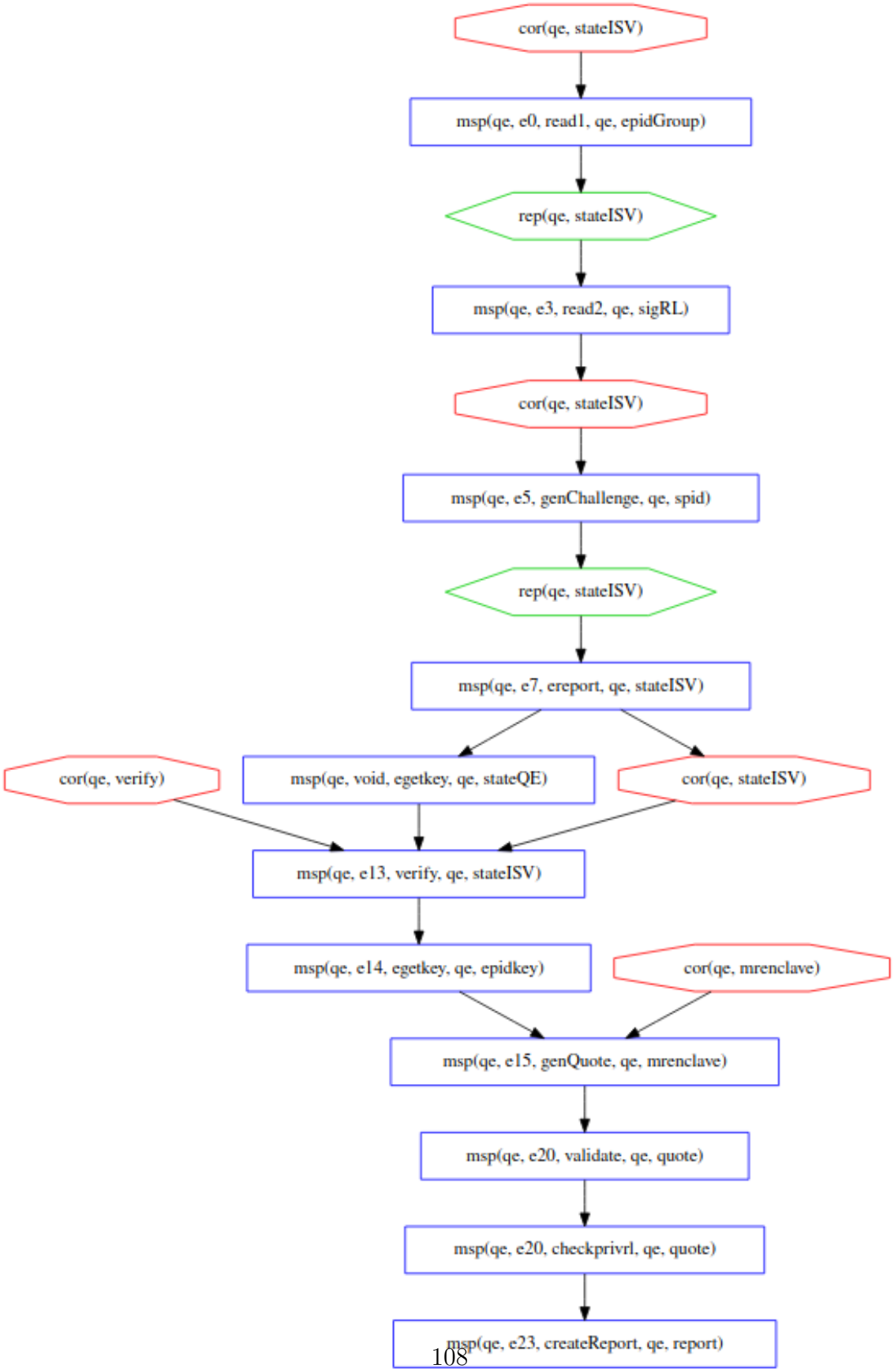

cor(qe, stateISV)
msp(qe, e0, read1, qe, epidGroup)
rep(qe, stateISV)
msp(qe, e3, read2, qe, sigRL)
cor(qe, stateISV)
msp(sv, e5, genChallenge, sv, spid)
cor(qe, c)
msp(qe, e7, ereport, qe, stateISV)
msp(qe, void, egetkey, qe, stateQE)
cor(qe, verify)
msp(qe, e13, verify, qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(qe, e20, validate, qe, quote)
msp(qe, e20, checkprivrl, qe, quote)
msp(qe, e23, createReport, qe, report)

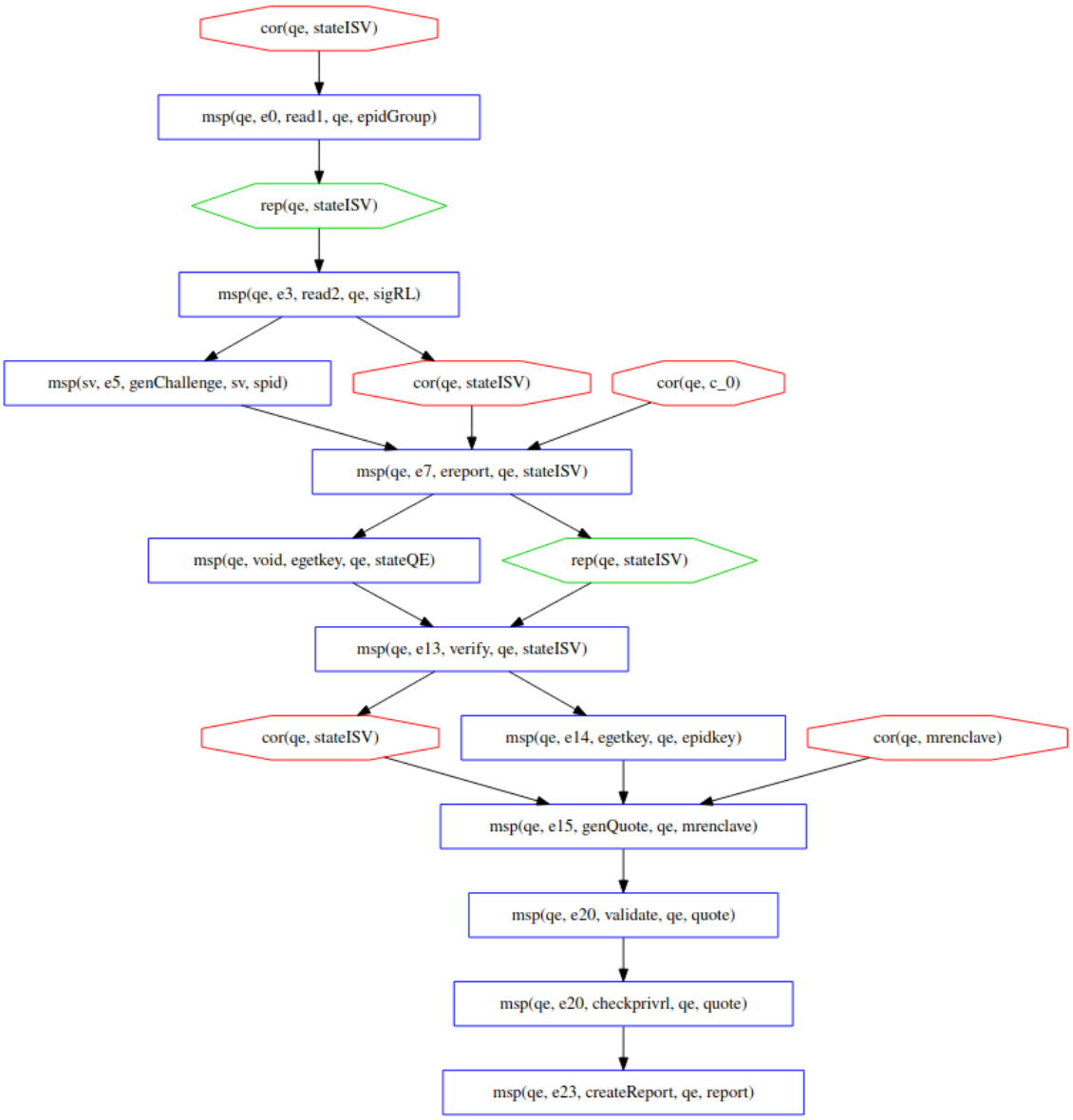

cor(qe, stateISV)
msp(qe, e0, read1, qe, epidGroup)
msp(ias, e3, read2, ias, sigRL)
rep(qe, stateISV)
msp(qe, e5, genChallenge, qe, spid)
cor(qe, stateISV)
cor(qe, c)
msp(qe, e7, ereport, qe, stateISV)
msp(qe, void, egetkey, qe, stateQE)
cor(qe, verify)
msp(qe, e13, verify, qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(ias, e20, validate, ias, quote)
msp(ias, e20, checkprivrl, ias, quote)
msp(ias, e23, createReport, ias, report)

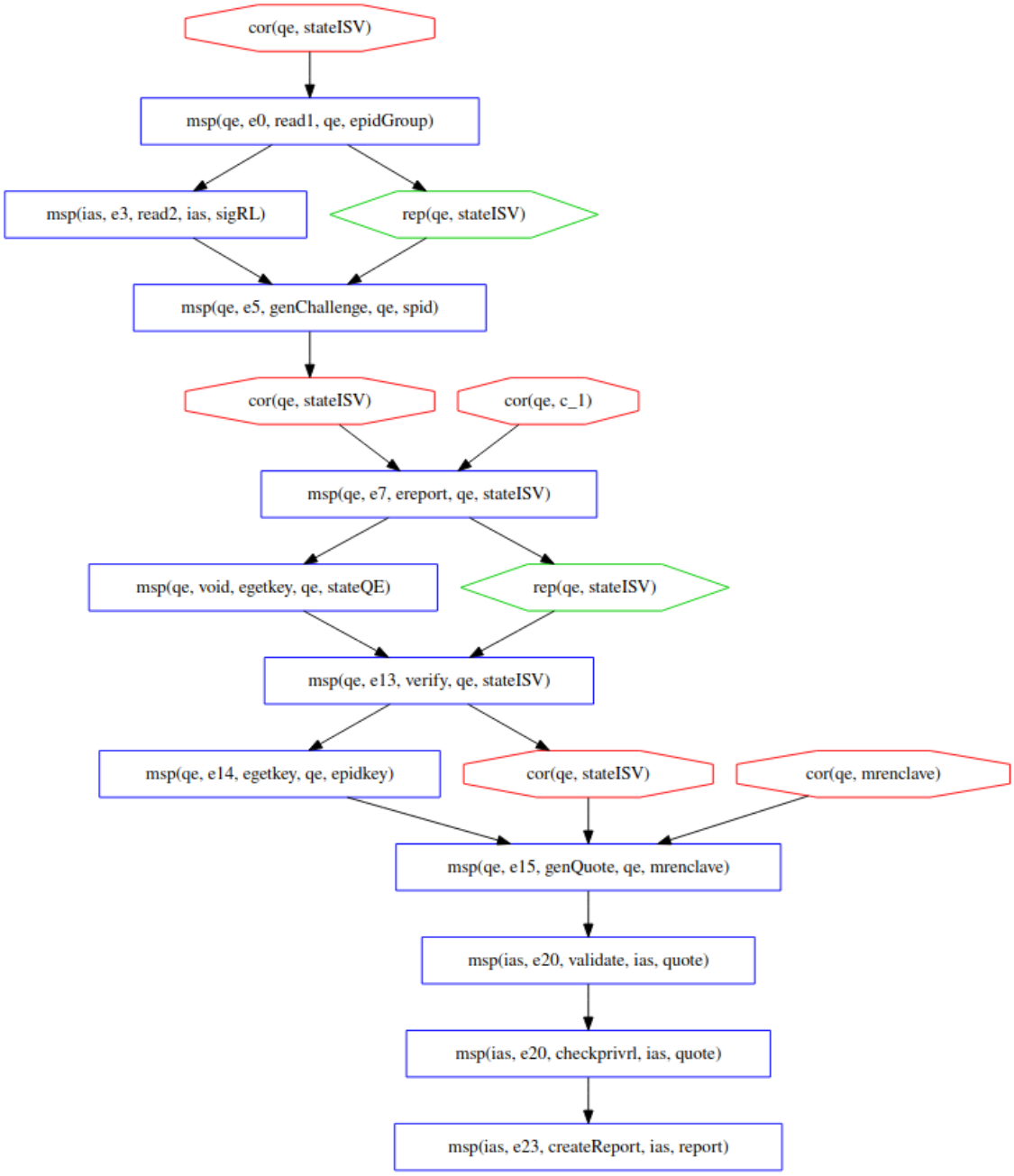

cor(qe, stateISV)
msp(qe, e0, read1, qe, epidGroup)
msp(qe, e3, read2, qe, sigRL)
rep(qe, stateISV)
msp(qe, e5, genChallenge, qe, spid)
cor(qe, stateISV)
cor(qe, c)
msp(qe, e7, ereport, qe, stateISV)
msp(qe, void, egetkey, qe, stateQE)
cor(qe, verify)
msp(qe, e13, verify, qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(qe, e20, validate, qe, quote)
msp(qe, e20, checkprivrl, qe, quote)
msp(qe, e23, createReport, qe, report)

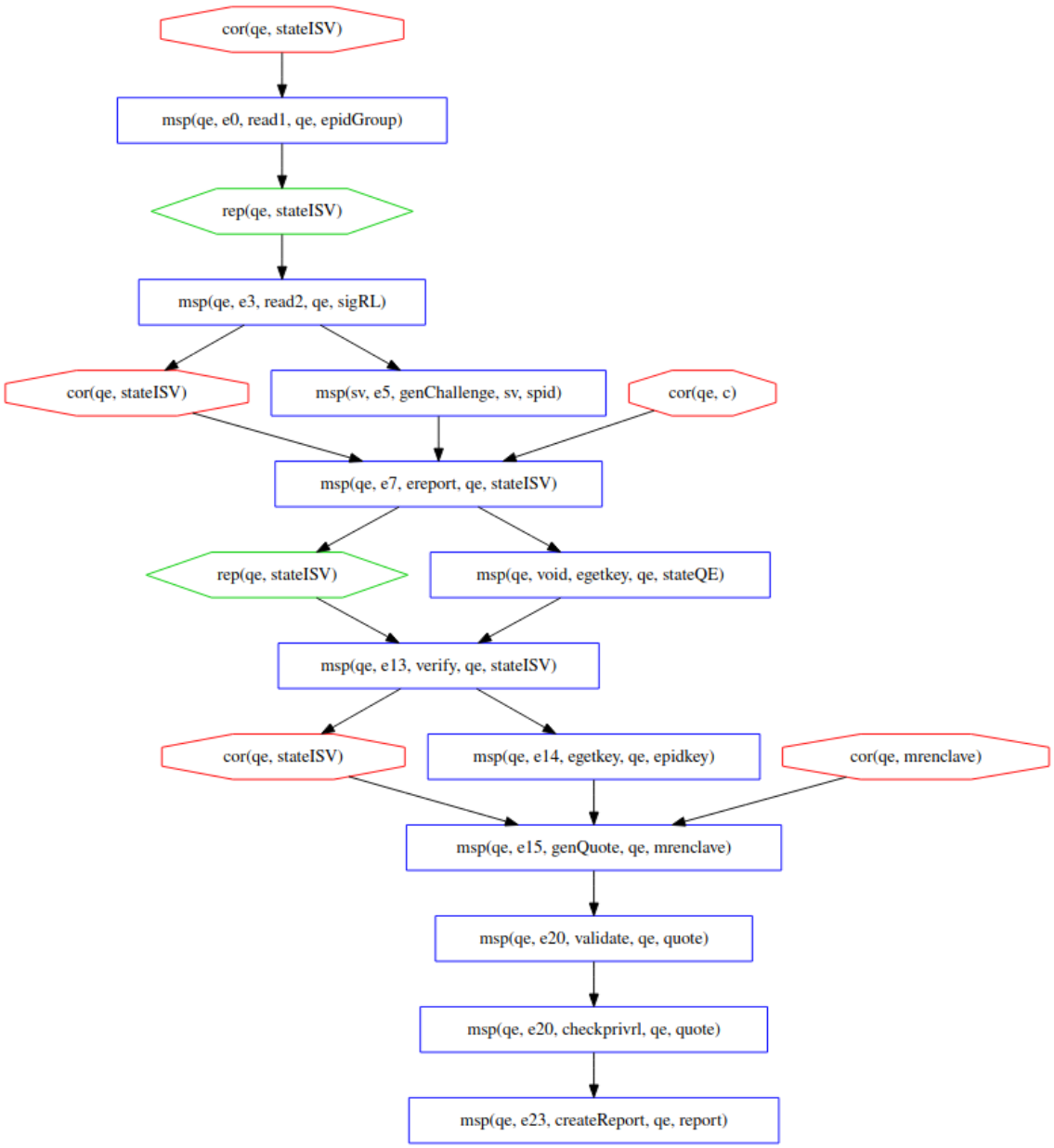

cor(qe, stateISV)
msp(qe, e0, read1, qe, epidGroup)
rep(qe, stateISV)
msp(qe, e3, read2, qe, sigRL)
cor(qe, stateISV)
msp(qe, e5, genChallenge, qe, spid)
rep(qe, stateISV)
msp(qe, e7, ereport, qe, stateISV)
msp(qe, void, egetkey, qe, stateQE)
msp(qe, e13, verify, qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, stateISV)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(qe, e20, validate, qe, quote)
msp(qe, e20, checkprivrl, qe, quote)
msp(qe, e23, createReport, qe, report)

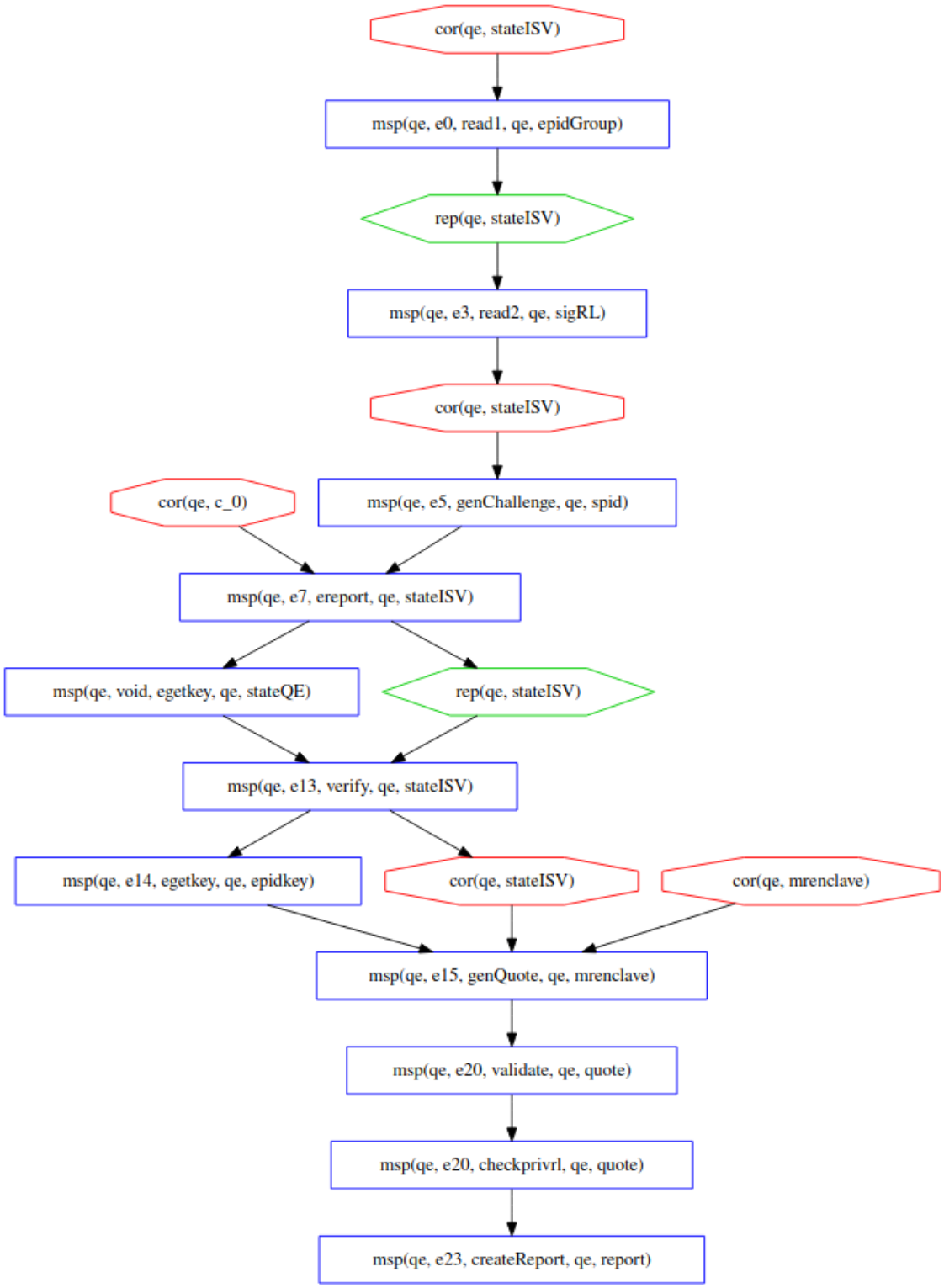

cor(qe, stateISV)
msp(qe, e0, read1, qe, epidGroup)
rep(qe, stateISV)
msp(qe, e3, read2, qe, sigRL)
cor(qe, stateISV)
msp(qe, e5, genChallenge, qe, spid)
rep(qe, stateISV)
msp(qe, e7, ereport, qe, stateISV)
msp(qe, void, egetkey, qe, stateQE)
msp(qe, e13, verify, qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, stateISV)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(qe, e20, validate, qe, quote)
msp(qe, e20, checkprivrl, qe, quote)
msp(qe, e23, createReport, qe, report)

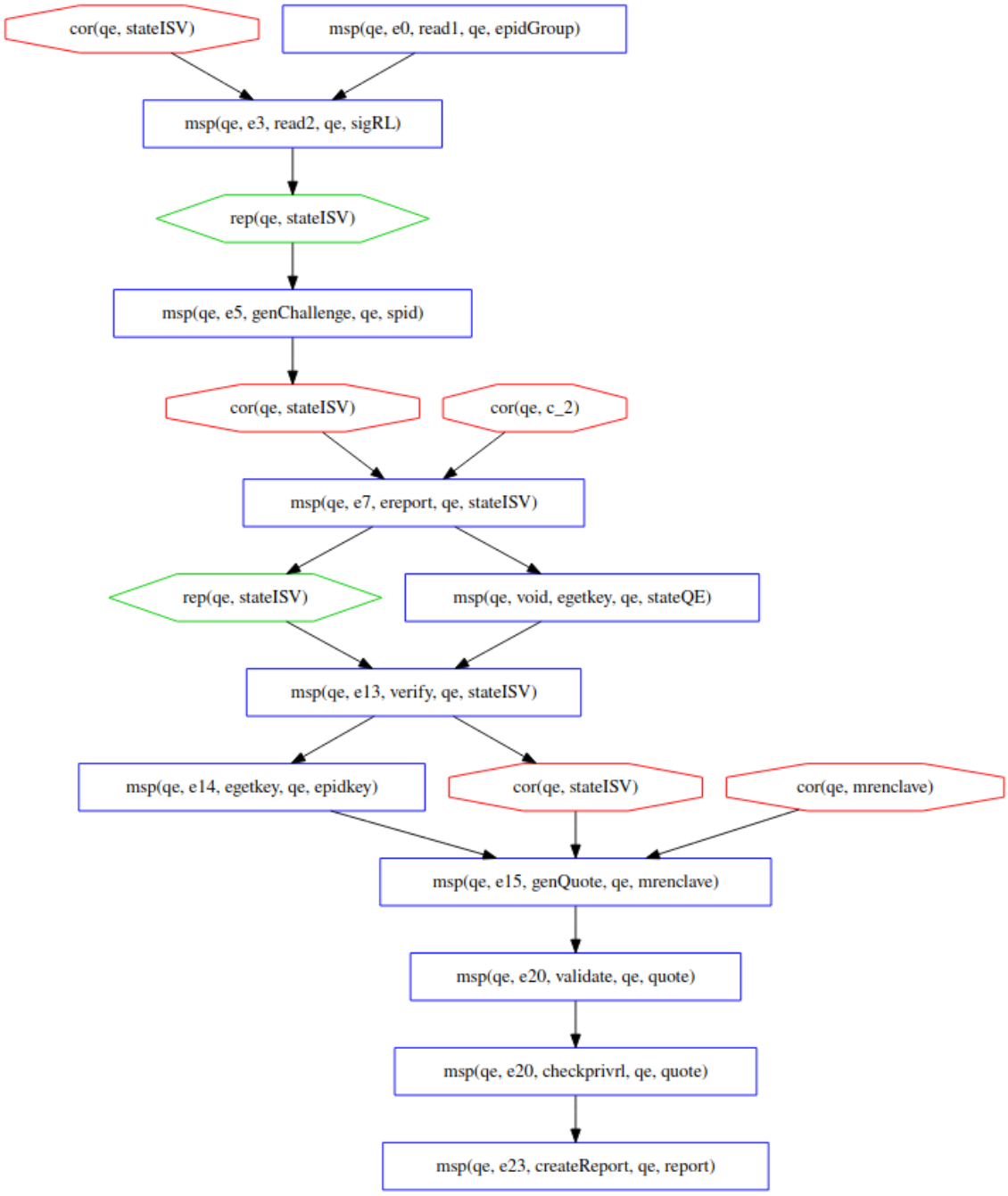

cor(qe, stateISV)
msp(qe, e0, read1, qe, epidGroup)
rep(qe, stateISV)
msp(qe, e3, read2, qe, sigRL)
cor(qe, stateISV)
msp(qe, e5, genChallenge, qe, spid)
rep(qe, stateISV)
msp(qe, e7, ereport, qe, stateISV)
cor(qe, verify)
msp(qe, void, egetkey, qe, stateQE)
cor(qe, stateISV)
msp(qe, e13, verify, qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(qe, e20, validate, qe, quote)
msp(qe, e20, checkprivrl, qe, quote)
msp(qe, e23, createReport, qe, report)

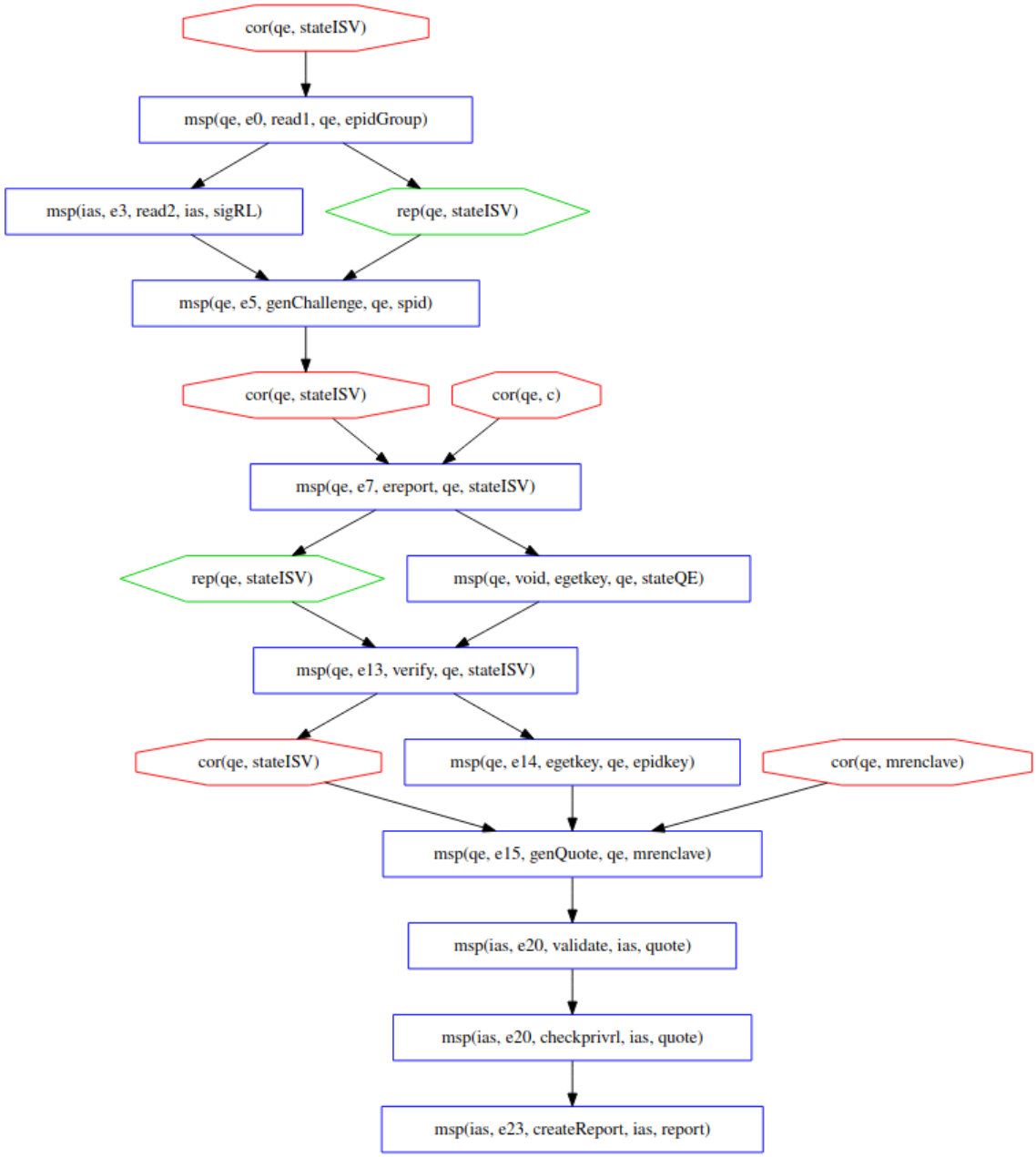

cor(qe, stateISV)
msp(qe, e0, read1, qe, epidGroup)
rep(qe, stateISV)
msp(qe, e3, read2, qe, sigRL)
msp(sv, e5, genChallenge, sv, spid)
cor(qe, stateISV)
cor(qe, c_0)
msp(qe, e7, ereport, qe, stateISV)
msp(qe, void, egetkey, qe, stateQE)
rep(qe, stateISV)
msp(qe, e13, verify, qe, stateISV)
cor(qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(qe, e20, validate, qe, quote)
msp(qe, e20, checkprivrl, qe, quote)
msp(qe, e23, createReport, qe, report)

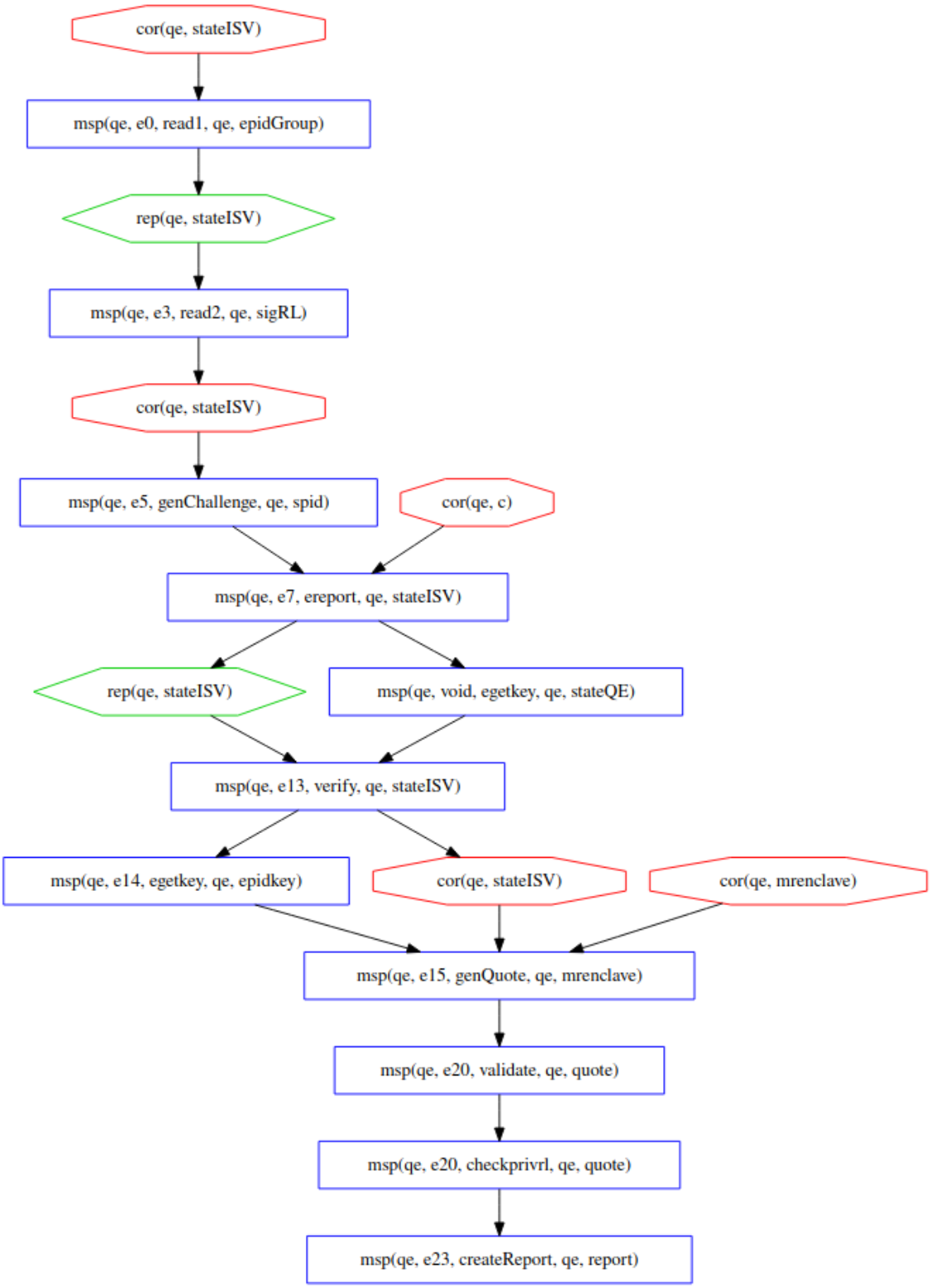

cor(qe, stateISV)
msp(qe, e0, read1, qe, epidGroup)
msp(ias, e3, read2, ias, sigRL)
rep(qe, stateISV)
msp(qe, e5, genChallenge, qe, spid)
cor(qe, stateISV)
cor(qe, c_1)
msp(qe, e7, ereport, qe, stateISV)
msp(qe, void, egetkey, qe, stateQE)
rep(qe, stateISV)
msp(qe, e13, verify, qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, stateISV)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(ias, e20, validate, ias, quote)
msp(ias, e20, checkprivrl, ias, quote)
msp(ias, e23, createReport, ias, report)

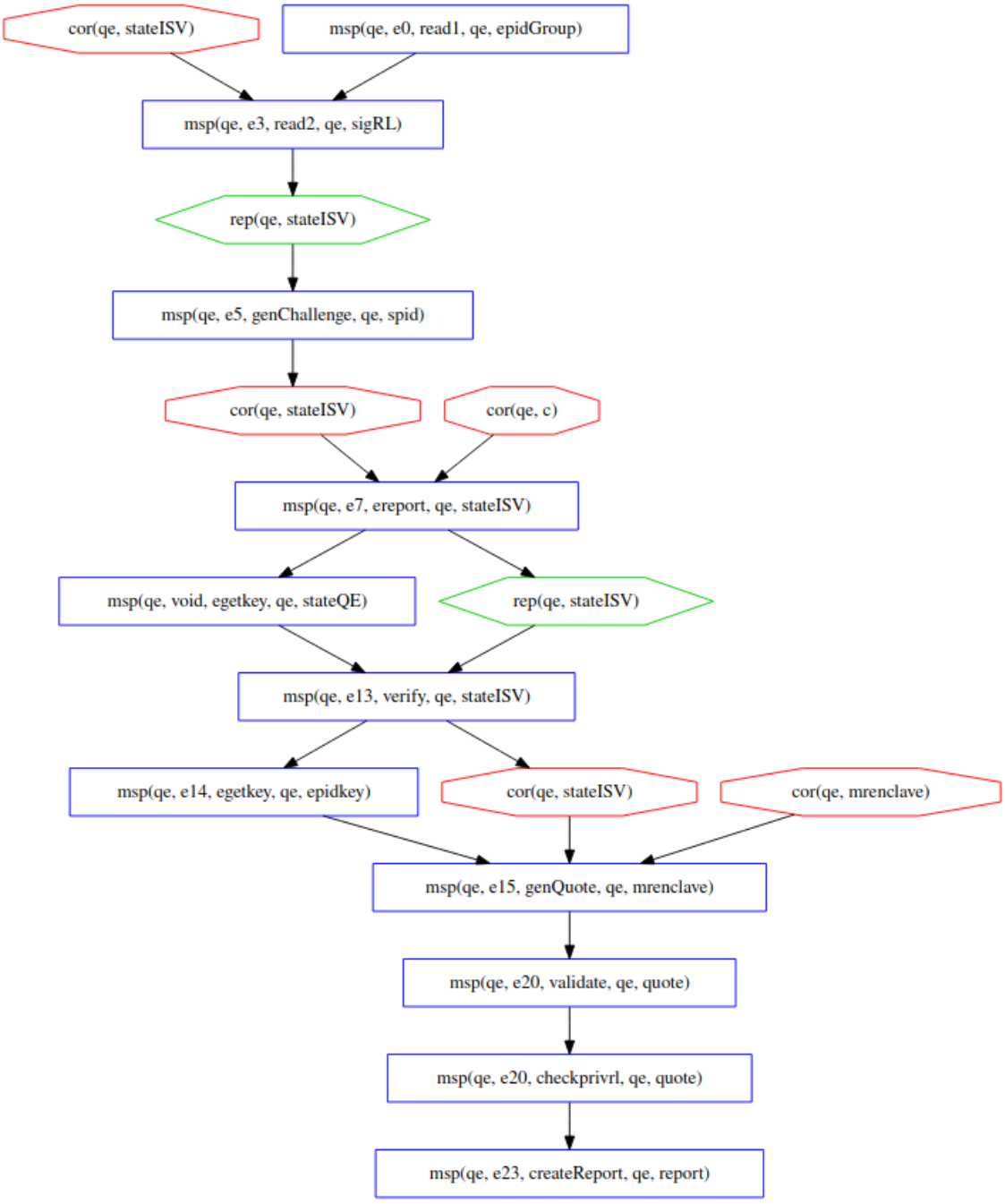

cor(qe, stateISV)
msp(qe, e0, read1, qe, epidGroup)
rep(qe, stateISV)
msp(qe, e3, read2, qe, sigRL)
cor(qe, stateISV)
msp(sv, e5, genChallenge, sv, spid)
cor(qe, c)
msp(qe, e7, ereport, qe, stateISV)
rep(qe, stateISV)
msp(qe, void, egetkey, qe, stateQE)
msp(qe, e13, verify, qe, stateISV)
cor(qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(qe, e20, validate, qe, quote)
msp(qe, e20, checkprivrl, qe, quote)
msp(qe, e23, createReport, qe, report)

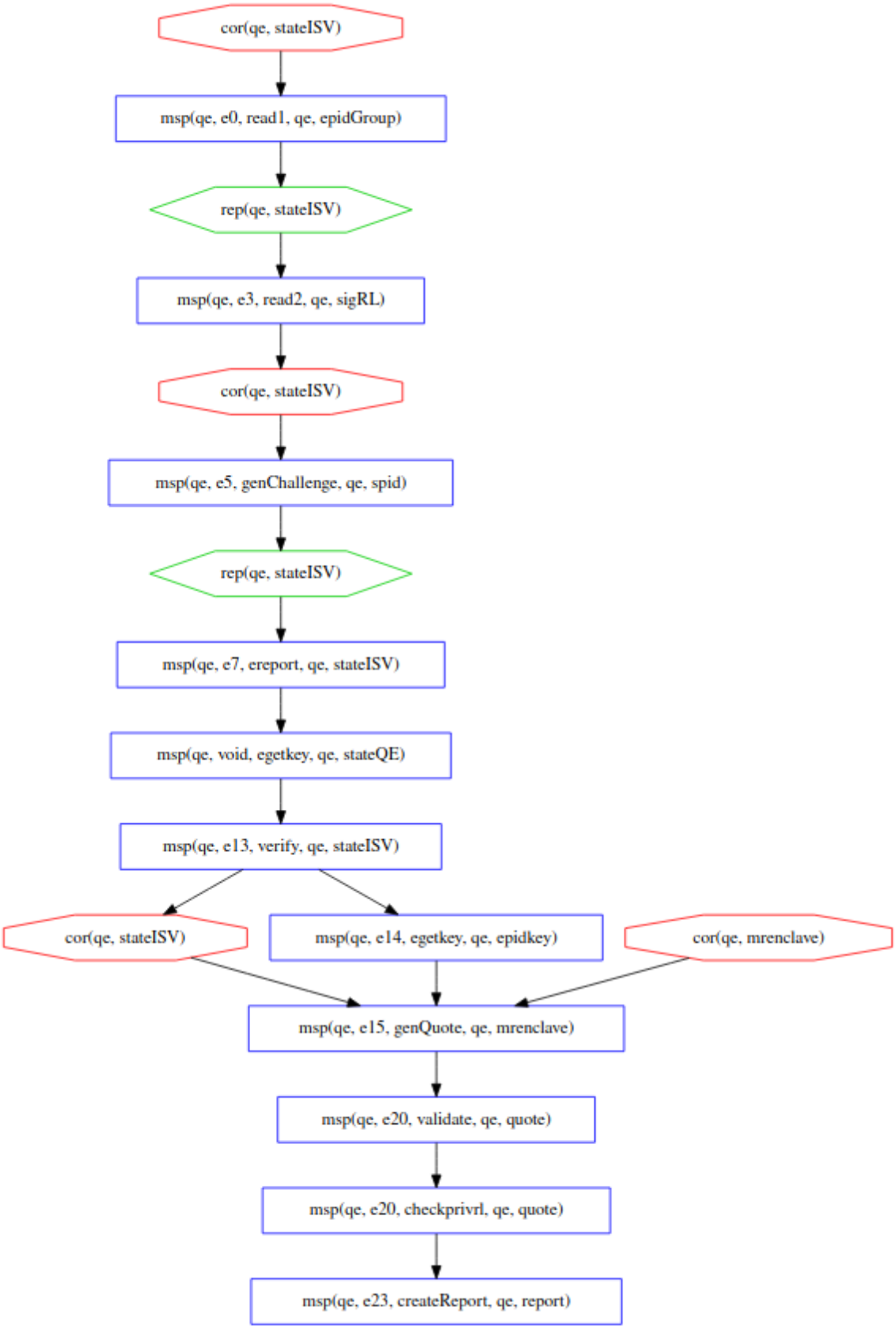

cor(qe, stateISV)
msp(qe, e0, read1, qe, epidGroup)
rep(qe, stateISV)
msp(qe, e3, read2, qe, sigRL)
cor(qe, stateISV)
cor(qe, c_0)
msp(qe, e5, genChallenge, qe, spid)
msp(qe, e7, ereport, qe, stateISV)
msp(qe, void, egetkey, qe, stateQE)
rep(qe, stateISV)
msp(qe, e13, verify, qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, stateISV)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(qe, e20, validate, qe, quote)
msp(qe, e20, checkprivrl, qe, quote)
msp(qe, e23, createReport, qe, report)

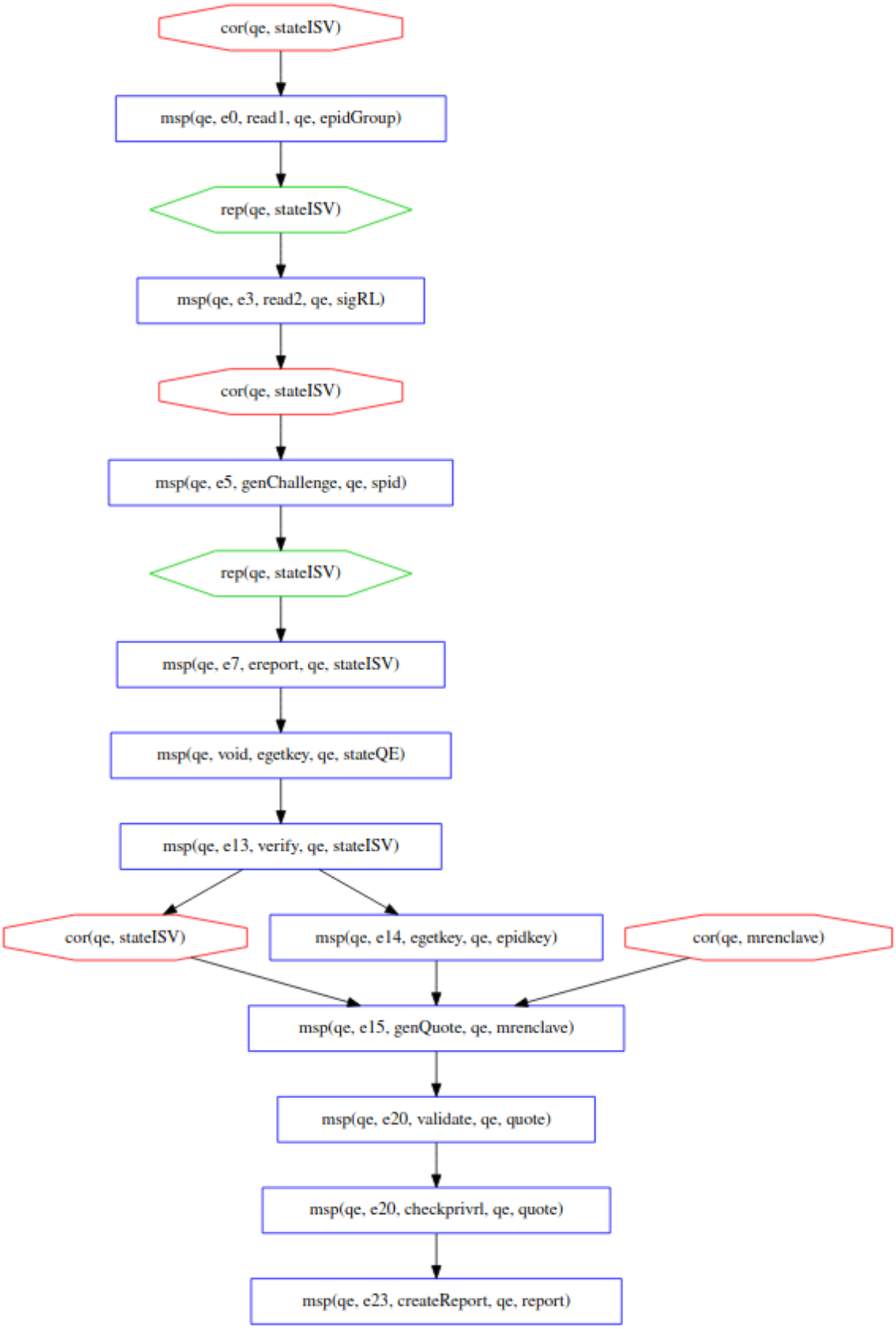

cor(qe, stateISV)
msp(qe, e0, read1, qe, epidGroup)
msp(qe, e3, read2, qe, sigRL)
rep(qe, stateISV)
msp(qe, e5, genChallenge, qe, spid)
cor(qe, stateISV)
cor(qe, c_2)
msp(qe, e7, ereport, qe, stateISV)
rep(qe, stateISV)
msp(qe, void, egetkey, qe, stateQE)
msp(qe, e13, verify, qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, stateISV)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(qe, e20, validate, qe, quote)
msp(qe, e20, checkprivrl, qe, quote)
msp(qe, e23, createReport, qe, report)

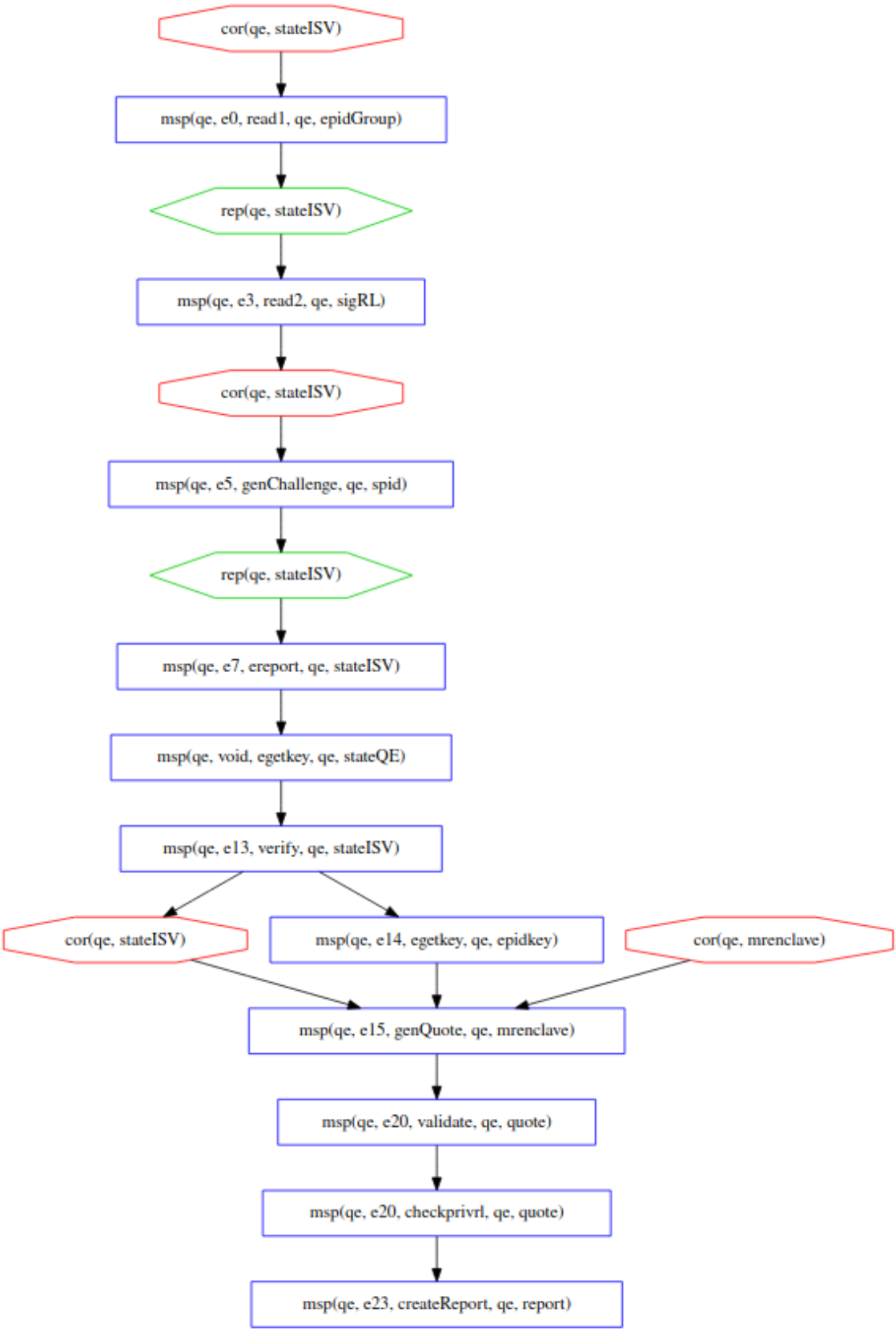

cor(qe, stateISV)
msp(qe, e0, read1, qe, epidGroup)
msp(ias, e3, read2, ias, sigRL)
rep(qe, stateISV)
msp(qe, e5, genChallenge, qe, spid)
cor(qe, stateISV)
cor(qe, c)
msp(qe, e7, ereport, qe, stateISV)
rep(qe, stateISV)
msp(qe, void, egetkey, qe, stateQE)
msp(qe, e13, verify, qe, stateISV)
cor(qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(ias, e20, validate, ias, quote)
msp(ias, e20, checkprivrl, ias, quote)
msp(ias, e23, createReport, ias, report)

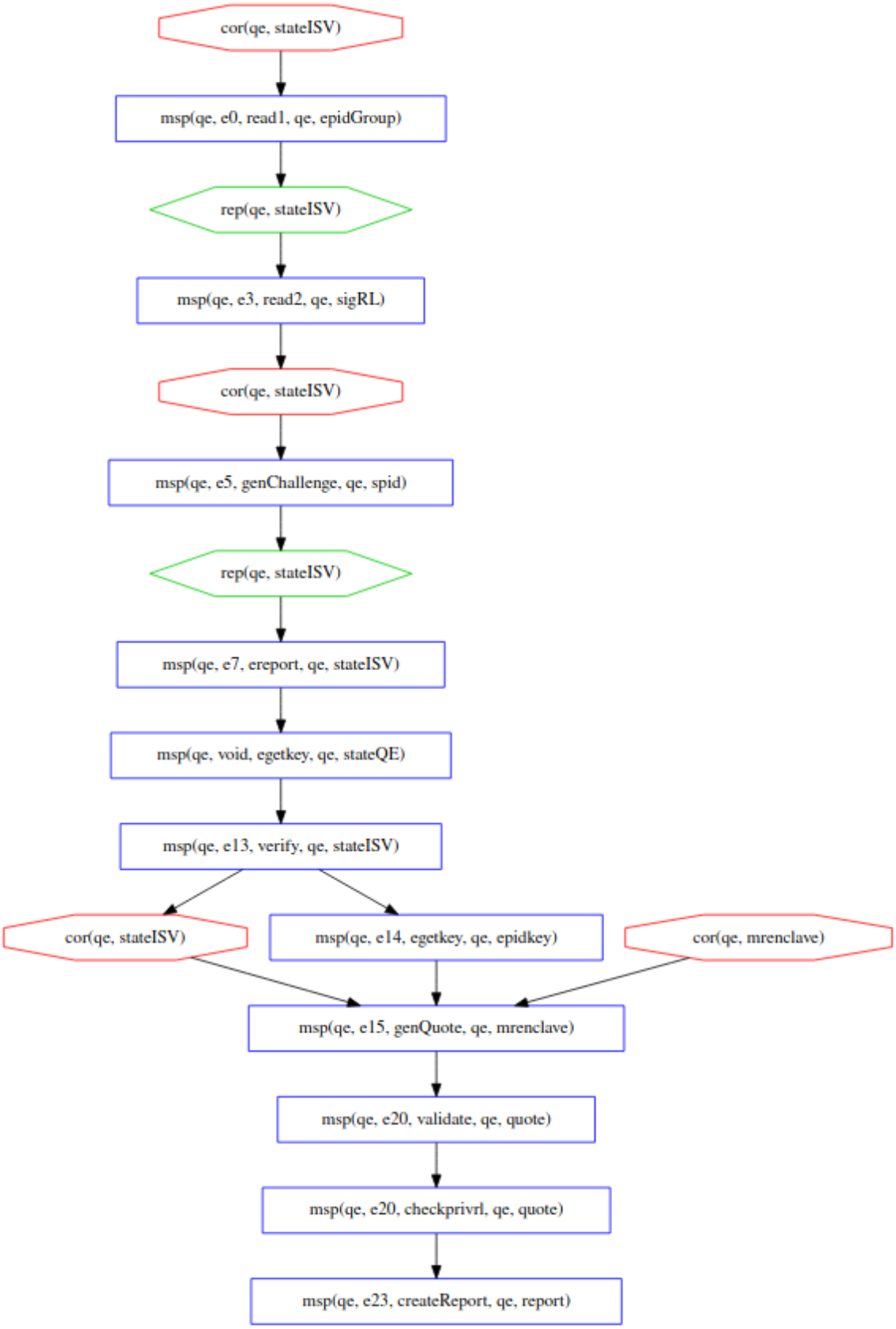

cor(qe, stateISV)
msp(qe, e0, read1, qe, epidGroup)
rep(qe, stateISV)
msp(qe, e3, read2, qe, sigRL)
cor(qe, stateISV)
msp(qe, e5, genChallenge, qe, spid)
cor(qe, c)
msp(qe, e7, ereport, qe, stateISV)
rep(qe, stateISV)
msp(qe, void, egetkey, qe, stateQE)
msp(qe, e13, verify, qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, stateISV)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(qe, e20, validate, qe, quote)
msp(qe, e20, checkprivrl, qe, quote)
msp(qe, e23, createReport, qe, report)

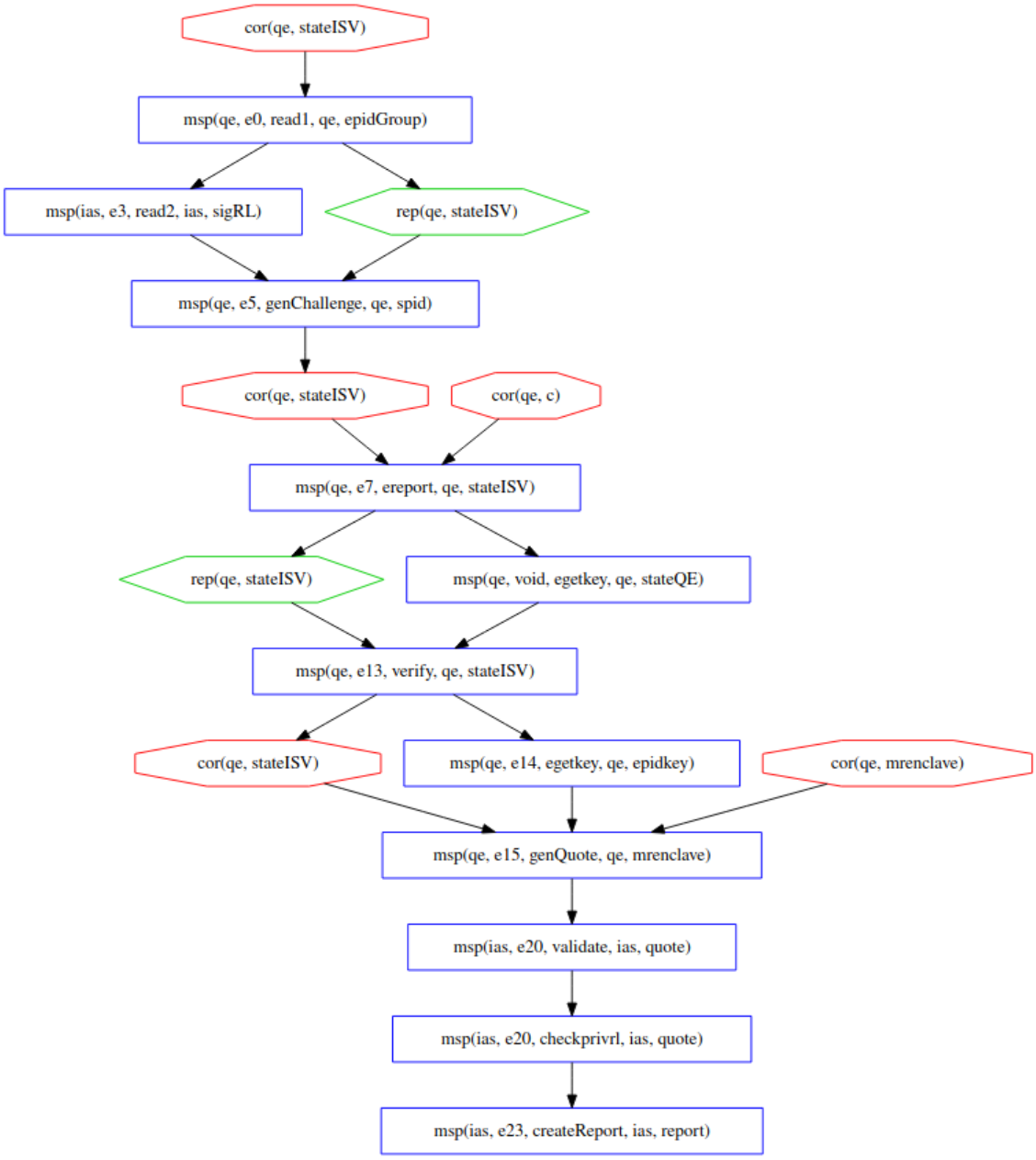

cor(qe, stateISV)
msp(qe, e0, read1, qe, epidGroup)
msp(qe, e3, read2, qe, sigRL)
rep(qe, stateISV)
msp(qe, e5, genChallenge, qe, spid)
cor(qe, stateISV)
cor(qe, c)
msp(qe, e7, ereport, qe, stateISV)
msp(qe, void, egetkey, qe, stateQE)
rep(qe, stateISV)
msp(qe, e13, verify, qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, stateISV)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(qe, e20, validate, qe, quote)
msp(qe, e20, checkprivrl, qe, quote)
msp(qe, e23, createReport, qe, report)

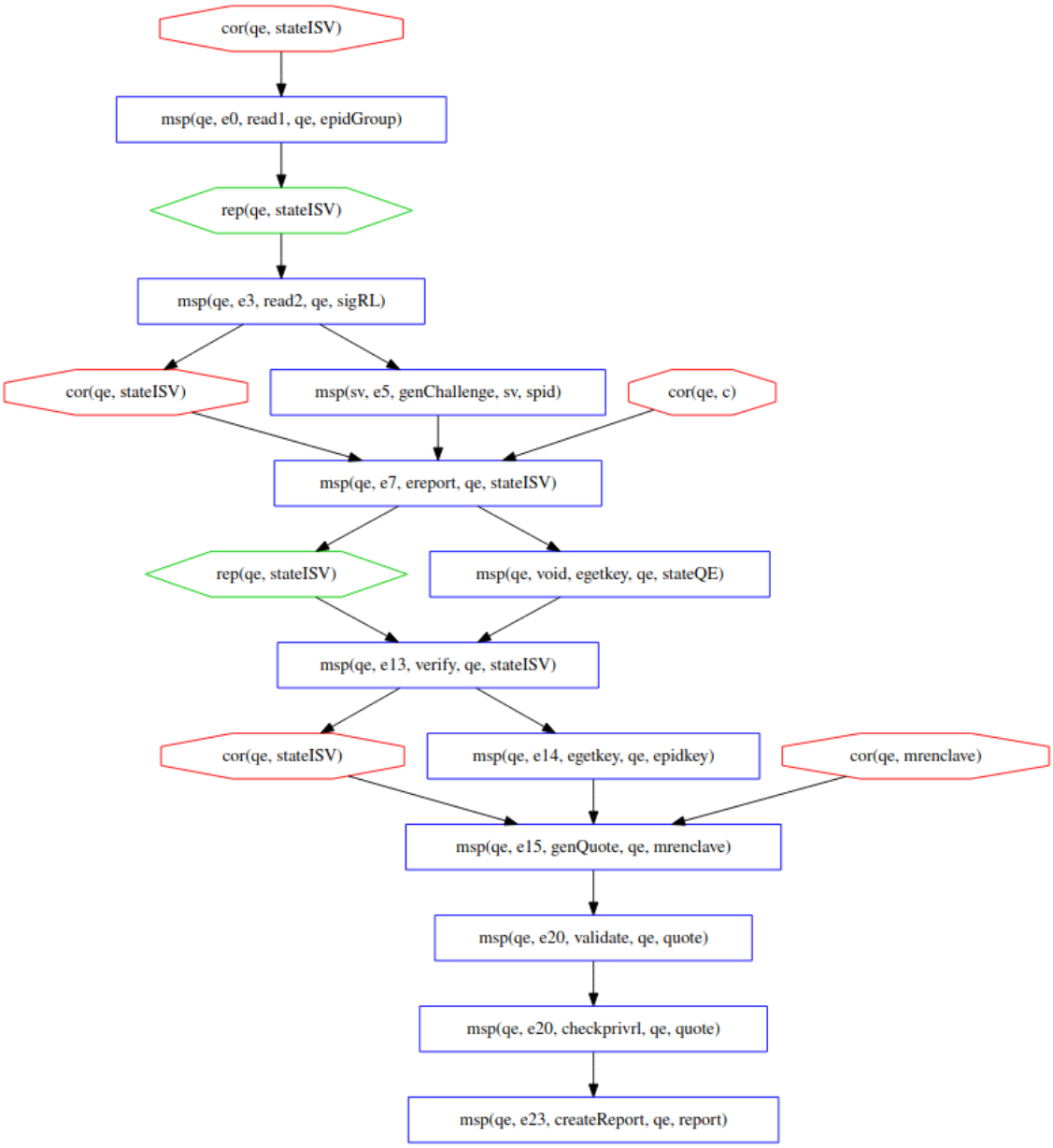

cor(qe, stateISV)
msp(qe, e0, read1, qe, epidGroup)
rep(qe, stateISV)
msp(qe, e3, read2, qe, sigRL)
cor(qe, stateISV)
msp(qe, e5, genChallenge, qe, spid)
rep(qe, stateISV)
msp(qe, e7, ereport, qe, stateISV)
msp(qe, void, egetkey, qe, stateQE)
msp(qe, e13, verify, qe, stateISV)
cor(qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(qe, e20, validate, qe, quote)
msp(qe, e20, checkprivrl, qe, quote)
msp(qe, e23, createReport, qe, report)

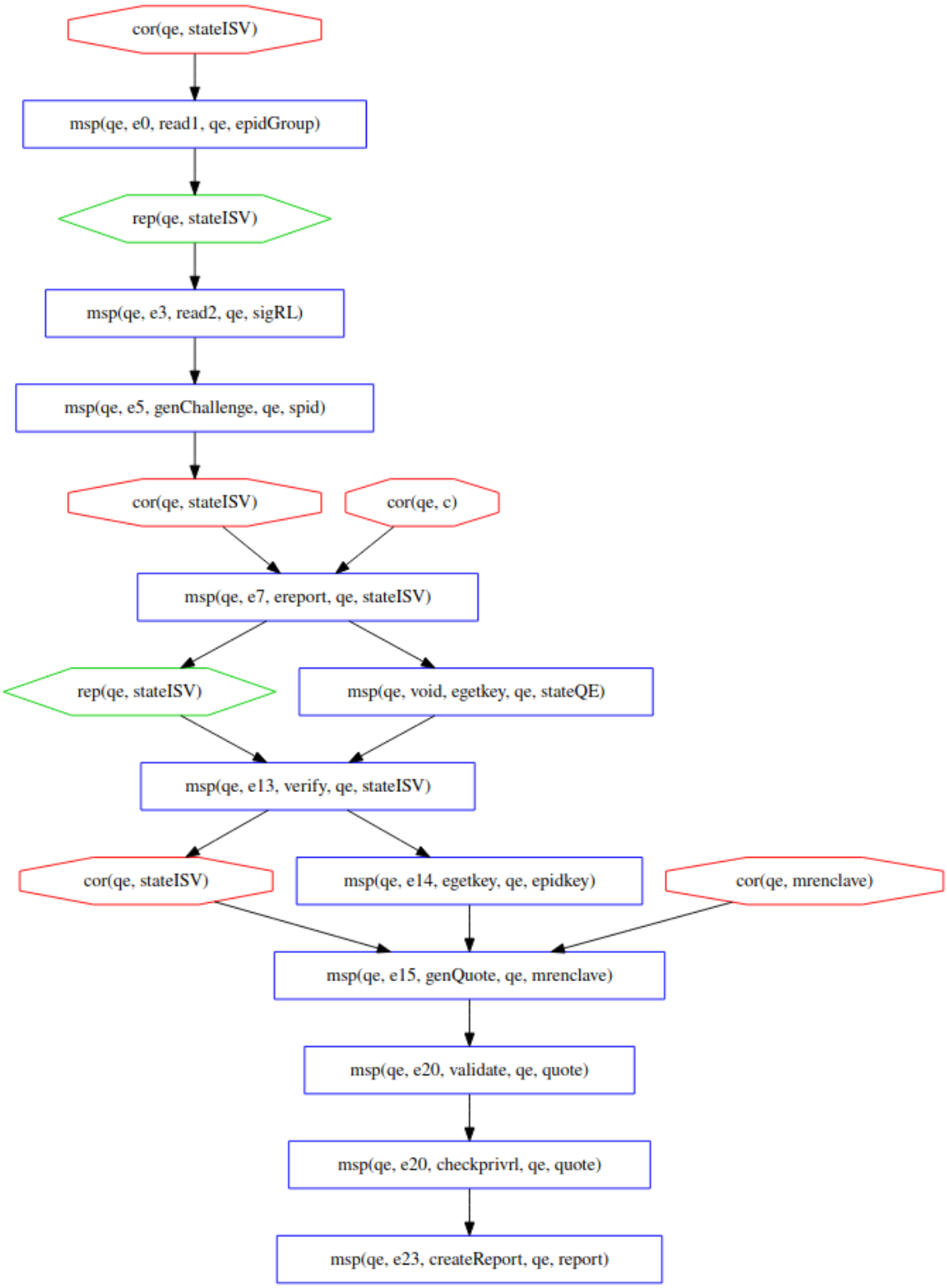

cor(qe, stateISV)
msp(qe, e0, read1, qe, epidGroup)
rep(qe, stateISV)
msp(qe, e3, read2, qe, sigRL)
cor(qe, stateISV)
msp(qe, e5, genChallenge, qe, spid)
rep(qe, stateISV)
msp(qe, e7, ereport, qe, stateISV)
msp(qe, void, egetkey, qe, stateQE)
msp(qe, e13, verify, qe, stateISV)
cor(qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(qe, e20, validate, qe, quote)
msp(qe, e20, checkprivrl, qe, quote)
msp(qe, e23, createReport, qe, report)


cor(qe, stateISV)
msp(qe, e0, read1, qe, epidGroup)
rep(qe, stateISV)
msp(qe, e3, read2, qe, sigRL)
cor(qe, stateISV)
msp(qe, e5, genChallenge, qe, spid)
rep(qe, stateISV)
msp(qe, e7, ereport, qe, stateISV)
msp(qe, void, egetkey, qe, stateQE)
msp(qe, e13, verify, qe, stateISV)
cor(qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(qe, e20, validate, qe, quote)
msp(qe, e20, checkprivrl, qe, quote)
msp(qe, e23, createReport, qe, report)


cor(qe, stateISV)
msp(qe, e0, read1, qe, epidGroup)
rep(qe, stateISV)
msp(qe, e3, read2, qe, sigRL)
cor(qe, stateISV)
msp(qe, e5, genChallenge, qe, spid)
rep(qe, stateISV)
msp(qe, e7, ereport, qe, stateISV)
msp(qe, void, egetkey, qe, stateQE)
msp(qe, e13, verify, qe, stateISV)
cor(qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(qe, e20, validate, qe, quote)
msp(qe, e20, checkprivrl, qe, quote)
msp(qe, e23, createReport, qe, report)


cor(qe, stateISV)
msp(qe, e0, read1, qe, epidGroup)
msp(ias, e3, read2, ias, sigRL)
rep(qe, stateISV)
msp(qe, e5, genChallenge, qe, spid)
cor(qe, stateISV)
cor(qe, c)
msp(qe, e7, ereport, qe, stateISV)
rep(qe, stateISV)
msp(qe, void, egetkey, qe, stateQE)
msp(qe, e13, verify, qe, stateISV)
cor(qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(ias, e20, validate, ias, quote)
msp(ias, e20, checkprivrl, ias, quote)
msp(ias, e23, createReport, ias, report)


cor(qe, stateISV)
msp(qe, e0, read1, qe, epidGroup)
rep(qe, stateISV)
msp(qe, e3, read2, qe, sigRL)
cor(qe, stateISV)
msp(sv, e5, genChallenge, sv, spid)
cor(qe, c)
msp(qe, e7, ereport, qe, stateISV)
rep(qe, stateISV)
msp(qe, void, egetkey, qe, stateQE)
msp(qe, e13, verify, qe, stateISV)
cor(qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(qe, e20, validate, qe, quote)
msp(qe, e20, checkprivrl, qe, quote)
msp(qe, e23, createReport, qe, report)


cor(qe, stateISV)
msp(qe, e0, read1, qe, epidGroup)
rep(qe, stateISV)
msp(qe, e3, read2, qe, sigRL)
msp(qe, e5, genChallenge, qe, spid)
cor(qe, stateISV)
cor(qe, c)
msp(qe, e7, ereport, qe, stateISV)
rep(qe, stateISV)
msp(qe, void, egetkey, qe, stateQE)
msp(qe, e13, verify, qe, stateISV)
cor(qe, stateISV)
msp(qe, e14, egetkey, qe, epidkey)
cor(qe, mrenclave)
msp(qe, e15, genQuote, qe, mrenclave)
msp(qe, e20, validate, qe, quote)
msp(qe, e20, checkprivrl, qe, quote)
msp(qe, e23, createReport, qe, report)