Towards Privacy-Preserving Fault Detection

Antonis Papadimitriou
University of Pennsylvania

Mingchen Zhao
University of Pennsylvania

Andreas Haeberlen
University of Pennsylvania

ABSTRACT

In this paper, we discuss the problem of detecting general faults in distributed systems that handle confidential information. Detecting non-crash faults is difficult in this setting because, to check the behavior of a given node, we need to know its expected behavior – but that can depend on the confidential information. Classical zero-knowledge proofs are difficult to apply because they are designed to verify functions with a fixed number of inputs, but in many distributed systems, both the size and the number of a node's "inputs" (the messages it has received from other nodes) are not known.

We propose an approach that can efficiently provide zero-knowledge fault detection for certain systems. Our approach spreads the detection tasks across multiple nodes, leveraging a node's existing knowledge whenever possible. We use epistemic reasoning to infer such knowledge, and we combine classical zero-knowledge proofs with a special data structure to handle inputs of unknown size. We show how our approach can be applied to a simple example system, and we report some initial performance measurements.

1. INTRODUCTION

Recent work on accountability [13, 16] has provided a way to detect a large class of non-crash faults in distributed systems. This seems useful: there are many faults – such as attacks by a malicious adversary, but also many software bugs, hardware malfunctions, or misconfigurations – that do not necessarily manifest as crash faults.

However, the ability to detect this more general class of faults comes at a price: systems like PeerReview [16] and AVM [13] require nodes to audit each other, and these audits reveal detailed information about each node's actions, including messages they have sent and received, and data they have stored or processed. *If the system handles private or confidential data, this may not be acceptable*: auditors in other administrative domains may not be authorized to see this data, and a compromised auditor could leverage the audit mechanism to extract information from other nodes. NetReview [14] and PVR [31] mitigate this problem in the context of BGP, but we are not aware of any protocol-independent fault detection techniques that can both handle non-crash faults and protect private data.

HotDep'13, November 3, 2013, Nemacolin Woodlands Resort, PA.

Accountability and privacy are not fundamentally incompatible; in principle, a node can use a zero-knowledge proof (ZKP) [10] to convince its auditors that it has correctly performed the functions it has been assigned. Recently there has been enormous progress towards practical ZKP techniques [9, 25]; however, direct application of ZKPs fails in our setting because of an "impedance mismatch" with distributed protocols. ZKPs generally assume that what needs to be verified is a *logic circuit* [30] – essentially a straight-line program, with fixed-size inputs and outputs and without unbounded loops or recursion – whereas the latter often specify each node as a *state machine* that sends and receives streams of messages to and from other nodes. Using logic circuits to verify the execution (or even the individual steps) of a state machine is difficult, and seems impossible in some cases, due to the problem of *structural privacy* [5]: if a node is invited to participate in verifying some transaction, it can inspect the circuit – its structure, the number and size of its inputs, the other nodes that are verifiers, etc. – and thus learn more about the transaction, even without seeing the actual data.

In this paper, we make a first step towards a practical, privacy-preserving fault detection mechanism for distributed systems. We do *not* propose a novel ZKP technique – rather, we argue that it may be sufficient to combine existing, circuit-based techniques with special-purpose data structures, such as Merkle Hash Trees [23], and to distribute the proof across multiple nodes to take advantage of the information the nodes already have. Using a simple example protocol, we demonstrate that it is possible to efficiently detect all observable faults (including non-crash faults) while maintaining very strong privacy guarantees. We believe that our approach can be extended to eventually cover a wide range of distributed protocols.

2. OVERVIEW

We consider a distributed system that consists of nodes N_1,N_2, N_3, ... Each node N_i has been assigned an algorithm A_i it should run – e.g., in the form of a state machine, whose inputs and outputs include user inputs, local outputs, and messages exchanged with other nodes. We say that N_i is *correct* in an execution of this system if it followed its algorithm A_i; otherwise we say that N_i is *faulty*. Faulty nodes are computationally bound (i.e., they cannot break cryptographic keys or invert hash functions), but can they otherwise be fully Byzantine [19], including collusion.

2.1 Goals

Our goal is to build a protocol that enables the correct nodes to identify faulty nodes. It is known [15] that not all faults are detectable in this setting; for instance, if a node corrupts its internal state but keeps sending the same messages as it

```
r1 joinReqD(@S,client,group)              :- joinReq(@client, group), S='server'.
r2 channelMember(@S,client,group,'YES')   :- joinReqD(@S, client, group),
                                             authorized(@S, group, client), S='server'.
r3 channelMember(@S,client,group,'NO')    :- joinReqD(@S, client, group),
                                             !authorized(@S, group, client), S='server'.
r4 joinResponse(@client,group,status)     :- channelMember(@S,client,group,status).
r5 allMsgs(@S,group,client,tstamp,msg)    :- message(@client, group, msg), tstamp = now(),
                                             channelMember(@S, client, group, 'YES').
r6 msgRcvd(@client,group,sender,ts,msg)   :- allMsgs(@S, group, sender, ts, msg),
                                             channelMember(@S, client, group, 'YES').
```

Figure 1: NDlog code for our running example – a simple chat server.

would have if it were correct, the other nodes cannot detect the fault. Hence, we can at most hope to detect *observable faults*; these are faults that (directly or transitively) affect at least one correct node.

We will make the simplifying assumption that the execution is finite, i.e., that the system has a *execution phase*, during which the original algorithm runs to completion, and a separate *detection phase*, during which the detection protocol runs. At the end of the detection phase, each node may report a set of other nodes on which it has detected a fault. The detection protocol should have the following three properties:

- **Completeness:** If a node N_i was observably faulty during the execution phase, at least one correct node N_j will report a fault on N_i.
- **Accuracy:** If node N_i was correct during the execution, no correct node will report a fault on N_i.
- **Privacy:** No node N_i can learn anything about a correct node $N_j \neq N_i$ during the detection phase that it could not already have learned during the execution phase, other than that N_j is correct.

We emphasize that faulty nodes are free to collude or misbehave during the detection phase if they wish, but that we do not require such faults to be reported.

2.2 Running example: Chat

For ease of exposition, we will explain our approach using a simple example protocol: a small chat server, whose code is shown in Figure 1. The code is written in Network Datalog [22], a declarative language. This is purely for ease of exposition; other languages, including imperative ones, could be used as well. We chose chat because it has the properties we discussed earlier: the number of groups, group members, and messages are all unbounded.

NDlog represents the state of a node a set of tables, which contain individual tuples; an NDlog program is basically a set of rules that describe how tuples can be derived from other tuples. For instance, the rule `X(@A,b,c) :- Y(@A,b,5), Z(@A,c)` says that a `(b,c)` tuple should be inserted into the X-table on node A whenever there is a `(b,5)` tuple in the Y-table and a `(c)` tuple in the Z-table. Rules may include tuples on different nodes; for instance, `W(@B,b,c) :- X(@A,b,c),b>5` says that A should send a message to B whenever a `(b,c)` tuple with $b > 5$ is inserted into its X-table, so that B can insert a `(b,c)` tuple into its W-table. b and c are variables that must be instantiated with concrete values when the rule is applied.

The chat server S has an `authorized` table that says which clients are allowed to join which groups, a `channelMember` table that says which clients are currently members of which groups, and an `allMsgs` table that contains all the posted messages. When a client C wants to join a group G, it inserts a `joinReq(@C,G)` tuple. This is sent to the server (r1), which must decide whether C is authorized to join G (r2+r3), update its `channelMember` table, and then send a `joinResponse` tuple back to C (r4). Once a member of G, C may post messages M by inserting a `message(@C,G,M)` tuple, which is sent to the server (r5) and forwarded to all the other members of the group (r6).

2.3 Minimum observer sets

In general, we can detect an observable fault on some node N as follows [15]: we collect a number of *observations* from correct nodes (messages these nodes did or did not send or receive), and we then establish that there is no execution of the system that is a) consistent with all observations, and in which b) N is correct.

For instance, suppose a client C_1 joins a group G, and another client C_2 then posts a message M to G. If the server S fails to forward M to C_1, S is clearly faulty – and this fault can be detected based on the observations of C_1 and C_2. The observations of these two nodes are sufficient (we do not need to ask any other nodes), and they are both necessary: if we know only that C_1 joined G, a plausible explanation is that S is correct and C_2 simply posted no messages, and if we know only that C_2 sent M, a plausible explanation is that S is correct and C_1 is not a member of G. We thus refer to $\{C_1, C_2\}$ as a *minimal observer set (MOS)* for this fault.

PeerReview [16] detects faults by collecting all observations about each node N_i at a central set of witness nodes. This simplifies audits considerably, but, as we have seen earlier, it violates N_i's privacy.

2.4 Approach

Our approach is based on the observation that fault detection does not *have* to be centralized: in principle, we can simply divide up the set of faults that could possibly occur on each node N_i, and we can assign the duty of checking for each fault to some of that fault's MOS. If the fault does occur, the nodes in the MOS will already have the necessary observations – *they do not need to exchange information with any other nodes.*

For verification, we distinguish three types of faults. *Simple faults (SF)* are ones whose MOS contains just a single node; that node can check for the fault directly. *Equivoca-*

tion faults (EF) occur when N_i pretends to some nodes in the MOS that some condition c is true, and to the others that c is false. EFs can be detected by requiring N_i to commit to c, and to reveal the commitment to the entire MOS. This ensures detection, and it does not violate privacy because both groups can already infer c from the N_i's actions. *Complex faults (CF)* are all remaining faults, of which more below.

Since the set of possible faults is typically infinite, we obviously need the ability to handle large groups of faults with a single proof. Intuitively, this can be done by considering ranges, such as the messages sent in a certain interval, or with a certain prefix; in Section 3.3, we describe a special data structure for this purpose. This trick should also help with detecting CFs: we can privately rule out most potential faults, and then fall back to running small circuit-based ZKPs to check for the (hopefully few) faults that remain plausible. For instance, if some chat messages had to pass through a special filter before being posted, we could first use commitments to determine the set of relevant messages, and then run a circuit for each message to check the filter.

Another concern is that nodes in the MOS could themselves be faulty. Some faults have more than one MOS,[1] so, if we expect up to f faults, we can simply assign each fault to f independent MOS, just as PeerReview would assign each node to f witnesses. But, somewhat counterintuitively, even if many (or all) faults have fewer than f different MOS, *this does not diminish the detection power of our approach*: if a fault can only be seen from $k < f$ groups of nodes, and each of these groups is compromised, then the fault is simply not observable and thus fundamentally cannot be detected in our setting [15], even with a non-private protocol.

3. CASE STUDY: CHAT

In this section, we illustrate how our approach can be applied to our running example, the chat server from Figure 1. We focus primarily on the server S, as it is more challenging to verify than the clients.

3.1 Deriving correctness conditions

First, we identify all the possible ways the server can fail. We observe that each rule τ`:-`α_1`,`α_2`,...` in Figure 1 implies two correctness conditions: a tuple must be derived whenever the preconditions hold ($\bigwedge \alpha_i \rightarrow \tau$), and second, the preconditions must have held whenever the tuple was derived ($\tau \rightarrow \bigwedge \alpha_i$). If there are other rules that could derive τ, the second condition becomes a disjunction of conjunctions, one for each such rule; if a rule contains free variables (as do all the rules in our example), we add a universal quantifier for that variable. The resulting set of conditions is complete: any observable fault must violate at least one of them.

3.2 Reasoning about knowledge

If $X \rightarrow Y$ is a correctness condition for S and there exists a single node N_i that can observe both X and Y, we are "done": we can simply ask N_i to verify that condition. But none of the conditions in our example are of that form; they all involve at least one (private) table on S. However, nodes can often infer at least some tuples that ought (not) to be in these tables. For instance, `joinReqD(C,G)` tuples can only be derived via `r1`, based on a `joinReq(G)` tuple C has sent to D. Thus, each client C can predict the state of `joinReqD(C,G)` on S based on the messages it has or has not sent to S. We can use this type of inference to expand the set of rules that a node can check.

We use *epistemic modal logic*, based on Halpern and Moses [18], to reason about the knowledge a given node N_i has about the state on S. In this logic, the operator K_i is used to indicate that a node N_i knows a certain fact. For instance, if τ `:-` α, β is the only rule that can derive τ, then we can make the inference $K_i\ \bar{\tau} \rightarrow K_i\ \bar{\alpha} \vee \bar{\beta}$, i.e., if node N_i knows that τ was not derived, it knows that either α or β must be absent. If N_i additionally knows that α is present, it can conclude that β is absent. By reasoning in this way, a client C can infer, for instance, that S must answer a `joinReq(C,G)` with a `joinResponse`: `r1` must trigger either `r2` or `r3`, depending on the state of `authorized(G,C)` (which C does not know), but, because of tertium non datur (the law of excluded middle), either of them must eventually trigger `r4`. We use a number of other inference rules, but we cannot present them here for lack of space.

For simple faults (SF), at least one client C will be able to infer both sides of the corresponding correctness condition in this way. For equivocation faults (EF), there will be multiple conditions that are each partially checkable by some client, except for some tuple(s) on S that none of the clients can infer. We can detect EFs by requiring S to *commit* to the state of these tuples, and to show that commitment to each of these clients.

3.3 Zero-knowledge maps

Since tables can be unbounded, we need a way to efficiently commit nodes to large ranges of tuples, without revealing the total size of the table. For this, we use a special kind of zero-knowledge set [24] we refer to as a *zero-knowledge map (ZKM)*. Suppose a node has a set of keys K and a mapping m that maps each $k \in K$ to some value $m(k)$. (Both keys and values can be arbitrary bitstrings.) Then the ZKM enables the node to produce a commitment to K and m such that, given a prefix p, the prover can selectively reveal to another node a) the set of keys that have the prefix p (if any), and b) the set of values that these keys map to, *without* revealing anything about other keys and values. If p has length zero, the prover fully reveals K and m.

We can implement a ZKM using a form of binary Merkle hash tree (MHT) [23] in which the edges from each parent vertex to its two children are labeled 0 and 1. Each key $k \in K$ is associated with the (unique) leaf L_k where the labels on the path from the root to L_k, concatenated, result in k. Next, we prune all vertices from the tree except for a) the leaves $L_k, k \in K$, b) any vertices on the path from a leaf L_k to the root, and c) any direct children of the vertices added under b). We then associate each remaining vertex with a hash value $H_i := H(s_i \,||\, H(e_i \,||\, v_i \,||\, r_i) \,||\, H_{i0} \,||\, H_{i1})$ where s_i is set to 1 if the vertex has any children, and 0 otherwise; e_i is 1 if $i \in K$, and 0 otherwise; v_i is $m(i)$ if $i \in K$, and empty otherwise; H_{i0} and H_{i1} are the hashes of the right and left children, if any, and empty otherwise;

[1] For instance, suppose node A is expected to receive numbers between 1 and 10 from B and C, add them, and send the sum to D and E. Suppose B and C send 2 and 3, respectively, and D and E each receive 18. Then there are four MOS: {B,D}, {C,E}, {B,E}, {C,D}.

```
c1a @C: ∀G: joinReq(@C,G) -> joinReqD(@S,C,G)
c1b @C: ∀G: !joinReq(@C,G) -> !joinReqD(@S,C,G)
c2a @C: ∀G: joinReq(@C,G) -> joinResponse(@C,G,'YES') ∨ joinResponse(@C,G,'NO')
c2b @C: ∀G: joinResponse(@C,G,'YES') ∨ joinResponse(@C,G,'NO') -> joinReq(@C,G)
c2c @C: ∀G: joinResponse(@C,G,'YES') -> channelMember(@S,C,G,'YES') ∧ authorized(@S,G,C)
c2d @C: ∀G: joinResponse(@C,G,'NO') -> channelMember(@S,C,G,'NO') ∧ !authorized(@S,G,C)
c2e @C: ∀G: !joinResponse(@C,G,'YES') ∧ !joinResponse(@C,G,'NO') -> !channelMember(@S,G,C,?)
c3a @C: ∀G∀T: joinResponse(@C,G,'YES') ∧ message(@C,G,M) -> allMsgs(@S,G,C,T,M)
c3b @C: ∀G∀T: !joinResponse(@C,G,'YES') ∨ !message(@C,G,M) -> !allMsgs(@S,G,C,T,M)
c4a @C: ∀G∀C'∀T: msgRcvd(@C,G,C',T,M) -> joinResponse(@C,G,'YES') ∧ channelMember(@S,C,G,YES)
c4b @C: ∀G∀C'∀T: joinResponse(@C,G,'YES') ∧ msgRcvd(@C,G,C',T,M) -> allMsgs(@S,G,C',T,M)
c4c @C: ∀G∀C'∀T: joinResponse(@C,G,'YES') ∧ msgRcvd(@C,G,C',T,M) -> channelMember(@S,C',G,YES)
c4d @C: ∀G∀C'∀T: joinResponse(@C,G,'YES') ∧ !msgRcvd(@C,G,C',T,M) -> !allMsgs(@S,G,C',T,M)
```

Figure 2: Conditions each client C verifies on the server S. Tuples in ZKMs are in bold; keys are underlined.

r_i is a random bitstring that is sufficiently long to prevent guessing; H is a cryptographic hash function that behaves as a random oracle (i.e., is both binding and hiding); and $||$ is concatenation. Then the commitment is the hash value that is assigned to the root of the MHT.

To produce a proof for a prefix p, the prover starts at the root and follows edges using the bits in p until a) all the bits in p have been used, or b) it reaches a leaf. In the first case, the prover reveals all the s_i, e_i, v_i, and r_i values in the subtree it has reached; in both cases, it reveals the s_i, $H(e_i \,||\, v_i \,||\, r_i)$, H_{i0}, and H_{i1} values along the path to the root. The verifier then checks whether the s_i bits are 1 for leaves and 0 for interior nodes, and it uses the provided values to recompute the hash value of the root. If the latter matches the commitment, the verifier accepts the proof.

3.4 Assigning keys

ZKMs enable us to efficiently check for equivocation faults: if a table T appears in conditions that are checked by different clients, we simply ask the server to put it into a ZKM, to commit to the ZKM, and to reveal to each client C the range of tuples in T that are visible to C. The key of the ZKM are the elements of T that are universally quantified in the conditions, in some order; the value are the elements of T that the client can see. Figure 2 show the conditions for our running example; the tables in ZKMs are in bold, and the keys are underlined. We have proven that these conditions, if checked by each client, satisfy all three of our requirements for a detection protocol (Section 2.1), but we omit the proof for lack of space.

An interesting challenge arises when choosing the order of the elements in the keys. For instance, at first glace, using C rather than $C \,||\, G$ as a key for `channelMember` seems equally plausible; however, notice that, in condition `c4c`, client C must verify the membership of some other client C' in a specific group G (to check whether the server may have fabricated a message). Since C may not know which other groups C' has joined, it is not safe to reveal the other memberships to C, so the ability to use prefixes of different lengths is necessary to preserve privacy.

3.5 Preliminary results

To test the efficiency of our detection protocol, we have implemented a checker for the conditions in Figure 2, based on ZKMs. We simulated a run with 200 clients and 50 groups; each client joined half of the groups at random and posted five messages per joined group. The probability of authorization was 0.5 per join request. We then ran the detection protocol. Generating the commitment for the largest ZKM ($\approx 637,000$ entries) took 2.41 s. The generation of a proof for each client took between 3.6 ms and 36 ms; the size of the proofs was between 639 kB and 2.4 MB, and the verification on the clients took between 28 ms and 113 ms. These results are initial evidence that efficient, privacy-preserving fault detection is possible, at least for some protocols.

4. CHALLENGES AND NEXT STEPS

In our current work, we are further generalizing our approach to broaden the range of protocols to which it can be applied. Below, we discuss some of the challenges and how we plan to address them.

Complex faults: Our running example contained only simple faults and equivocation faults. For complex faults, we can either use different data structures – such as the bit vector in PVR [31], which verifies a minimum over a set of integers, or techniques from privacy-preserving data mining [1, 20] – or fall back on general ZKP circuits. Compilers are available, e.g., in [3], that can generate such circuits from high-level programs. To preserve structural privacy, it is sufficient to ensure that all participating verifiers can statically predict a) the set of circuits they should participate in, and b) the structure of these circuits. We hope to accomplish this using the ZKMs and the epistemic reasoning from Sections 3.3 and 3.2, respectively.

Verification at runtime: In this paper, we made the simplifying assumption that the detection protocol runs after the execution has finished, but in practice, protocols can run for a long time. To enable verification of running protocols, we need to ensure that all verifiers are using a consistent snapshot of the prover's state and of their own relevant observations, perhaps using a variant of the Chandy-Lamport algorithm [4]; note, however, that the snapshot needs to cover just the union of the MOS and not the entire system. The addition of a privacy-preserving "checkpoint" should enable incremental checking, analogous to PeerReview [16].

Evidence and privacy: Accountability critically relies on the ability to prove the presence of a fault to a third party. In principle, evidence of faults could be constructed using a combination of signed messages and/or challenges that the faulty node cannot answer [16]; however, if privacy is an issue, *the evidence itself* may leak information! One way to mitigate this would be to shrink the evidence to the minimum that is necessary to prove the fault; another would be to verify the evidence itself in zero-knowledge.

5. RELATED WORK

The vision behind this work was described in an earlier position paper [17]. This paper contributes a concrete technical approach towards an efficient solution.

Verifiable computing: Recently there has been a lot of interest in efficient verification of outsourced computation, including PCPs [27], Pinocchio [25], Ginger [28], Zaatar [26], Allspice [29], and Gennaro et al. [9]; however, these focus on a two-party setting in which one party knows all the inputs, and are thus not directly applicable in our setting. Some special-case solutions for distributed protocols do exist: VEX [2] privately verifies ad exchanges, and our earlier work on PVR [12, 31] verifies BGP routing decisions.

Zero-knowledge proofs: There are numerous specialized zero-knowledge arguments, e.g., for set membership [24] or equality testing [8, 21], but these approaches are not general enough for our purposes. Lately, new techniques have been developed that can verify arbitrary computations, modeled as circuits [9, 11], but are less efficient. Our goal is to adapt both approaches to the context of distributed computing, and to combine them in a way that harnesses both efficiency and generality.

Privacy-preserving data mining: This line of work, initiated by [1, 20], focuses on extracting information from distributed, private data sets. It is typically optimized for very specific classes of algorithms that are relevant for data mining; for instance, P4P [6, 7] implements algorithms that can be decomposed into a sequence of vector additions. Thus, it is complementary to our own work, which focuses on general message-passing algorithms.

6. CONCLUSION

At first glance, privacy and accountability may appear to be incompatible: to detect whether a node has correctly performed the function it has been assigned, it seems necessary to know the node's actions. However, there is mounting evidence that privacy-preserving accountability is not only possible, but perhaps not even particularly expensive – neither in terms of overhead nor in terms of detection power!

The technique we present here is not yet a complete solution: it is still quite easy to find protocols for which it fails. However, we believe that it can be further generalized by integrating small ZKPs to detect complex faults; we are investigating this in our ongoing work.

Acknowledgments

We thank Ariel Feldman and Wenchao Zhou for helpful comments on an earlier draft of this paper. This work was supported by DARPA contract FA8650-11-C-7189, NSF grants CNS-1054229 and CNS-1065130, and by a gift from Google. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] R. Agrawal and R. Srikant. Privacy-preserving data mining. In *Proc. SIGMOD*, 2000.
[2] S. Angel and M. Walfish. Verifiable auctions for online ad exchanges. In *Proc. SIGCOMM*, Aug. 2013.
[3] M. Backes, M. Maffei, and K. Pecina. Automated synthesis of privacy-preserving distributed applications. In *NDSS*, 2012.
[4] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, Feb. 1985.
[5] S. Davidson, S. Khanna, S. Roy, J. Stoyanovich, V. Tannen, and Y. Chen. On provenance and privacy. In *ICDT*, 2011.
[6] Y. Duan, J. Canny, and J. Zhan. P4P: practical large-scale privacy-preserving distributed computation robust against malicious users. In *Proc. USENIX Security*, 2010.
[7] Y. Duan and J. F. Canny. Practical private computation and zero-knowledge tools for privacy-preserving distributed data mining. In *Proc. SDM*, 2008.
[8] R. Fagin, M. Naor, and P. Winkler. Comparing information without leaking it. *Commun. ACM*, 39(5):77–85, 1996.
[9] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *Proc. EUROCRYPT*, 2013.
[10] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof-systems. In *STOC*, 1985.
[11] J. Groth. Short pairing-based non-interactive zero-knowledge arguments. In *Proc. ASIACRYPT*, 2010.
[12] A. J. T. Gurney, A. Haeberlen, W. Zhou, M. Sherr, and B. T. Loo. Having your cake and eating it too: Routing security with privacy protections. In *Proc. HotNets*, Nov. 2011.
[13] A. Haeberlen, P. Aditya, R. Rodrigues, and P. Druschel. Accountable virtual machines. In *Proc. OSDI*, Oct. 2010.
[14] A. Haeberlen, I. Avramopoulos, J. Rexford, and P. Druschel. NetReview: Detecting when interdomain routing goes wrong. In *Proc. NSDI*, Apr 2009.
[15] A. Haeberlen and P. Kuznetsov. The Fault Detection Problem. In *Proc. OPODIS*, Dec. 2009.
[16] A. Haeberlen, P. Kuznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *Proc. SOSP*, Oct 2007.
[17] A. Haeberlen, M. Zhao, W. Zhou, A. J. T. Gurney, M. Sherr, and B. T. Loo. Privacy-preserving collaborative verification protocols. In *Proc. LADIS*, 2012.
[18] J. Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. *J. ACM*, 37(3):549–587, July 1990.
[19] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM TOPLAS*, 4(3):382–401, 1982.
[20] Y. Lindell and B. Pinkas. Privacy preserving data mining. In *Proc. CRYPTO*, 2000.
[21] H. Lipmaa. Verifiable homomorphic oblivious transfer and private equality test. In *Proc. ASIACRYPT*, 2003.
[22] B. Loo, T. Condie, M. Garofalakis, D. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking. *CACM*, 52(11):87–95, Nov. 2009.
[23] R. Merkle. Protocols for public key cryptosystems. In *Proc. Symposium on Security and Privacy*, Apr. 1980.
[24] S. Micali, M. Rabin, and J. Kilian. Zero-knowledge sets. In *Proc. FOCS*, Oct. 2003.
[25] B. Parno, C. Gentry, J. Howell, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *Proc. IEEE Symposium on Security and Privacy*, May 2013.
[26] S. Setty, B. Braun, V. Vu, A. J. Blumberg, B. Parno, and M. Walfish. Resolving the conflict between generality and plausibility in verified computation. In *Proc. EuroSys*, 2013.
[27] S. Setty, R. McPherson, A. J. Blumberg, and M. Walfish. Making argument systems for outsourced computation practical (sometimes). In *Proc. NDSS*, Feb. 2012.
[28] S. Setty, V. Vu, N. Panpalia, B. Braun, A. J. Blumberg, and M. Walfish. Taking proof-based verified computation a few steps closer to practicality. In *Proc. USENIX Security*, 2012.
[29] V. Vu, S. Setty, A. J. Blumberg, and M. Walfish. A hybrid architecture for interactive verifiable computation. In *Proc. IEEE Symposium on Security and Privacy*, May 2013.
[30] A. C. Yao. Protocols for secure computations. In *Proc. Symposium on Foundations of Computer Science (SFCS)*, 1982.
[31] M. Zhao, W. Zhou, A. J. T. Gurney, A. Haeberlen, M. Sherr, and B. T. Loo. Private and verifiable interdomain routing decisions. In *Proc. SIGCOMM*, 2012.