

Managing the Health of Security Experiments *

Jelena Mirkovic
USC Information Sciences
Institute
4676 Admiralty Way ste 1001
Marina Del Rey, CA 90292
sunshine@isi.edu

Karen Sollins
MIT CSAIL
32 Vassar St.
Cambridge, MA 02139
sollins@csail.mit.edu

John Wroclawski
USC Information Sciences
Institute
4676 Admiralty Way ste 1001
Marina Del Rey, CA 90292
jtw@isi.edu

ABSTRACT

Testbed experiments are a challenge to manage manually, because they involve multiple machines and their correctness depends on the correct operation of testbed infrastructure that is often hidden from the experimenter. Testbed experiments that recreate security events add management challenges of scale – they are often very large; complexity – many threats work only if certain conditions are met by the network environment; and risk – they often involve malicious code and disruptive actions that must be contained. Finally, an experiment may be run by someone who did not create it originally. It is challenging for this new experimenter to ascertain if any experiment behavior was intended or a sign of failure, and to diagnose and correct failures.

We introduce a new paradigm of *experiment health* that denotes a user-supplied description of correct experiment behavior, i.e., healthy experiments behave as their creators intended. We then propose an experiment health management infrastructure that can be added to existing testbeds to improve their usability and robustness. The infrastructure consists of an *expectation language* in which a user expresses her notion of experiment health, a monitoring infrastructure that is driven by user expectations, health evaluators, recovery engines and a shared library of health tools and collected experiment statistics. This infrastructure is useful not only for experiment management, but also for testbed management.

1. MOTIVATION

In the past decade several large and many small testbeds have been created to promote realistic testing of solutions in fields such as networking, distributed systems and applications, security, sensor networks, etc. Experiments on these testbeds are distributed by nature, which makes their monitoring, problem detection, diagnosis and recovery challenging if done in manual fashion. We refer to all these actions as experiment health management, where a healthy experiment behaves as its creator intended, *at all levels of observation*.

*This material is based upon work supported by the Department of Homeland Security, and Space and Naval Warfare Systems Center, San Diego, under Contract No. N66001-07-C-2001. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Department of Homeland Security for the Space and Naval Warfare Systems Center, San Diego.

This notion of health at all levels is necessary to differentiate a healthy experiment from an experiment that reaches a creator-intended goal but in a way that creator did not intend. We will illustrate this point in the following section.

Security experiments pose additional management challenges of high scale, complexity and risk. Because security events, such as DDoS or worm propagation, are high-scale, experiments that recreate them usually require tens to hundreds of machines, which increases cost and complexity of health management, and probability of health failures. Security experiments must recreate realistic threats to either study them or test proposed defenses. This is complex because security threats often require a complex set of conditions to be effective, such as presence of a particular vulnerable version of a specific application for a worm to propagate. Security defense performance also often depends on the deployment environment. If any puzzle piece is not set just right the threat or its defense may not succeed. Security experiments are often risky because they involve malware or create disruptive events. Testbeds need to effectively manage this risk, which is often done assuming user collaboration, e.g., requiring (but not checking or enforcing) that a user restricts malware probes only to machines within her experiment.

Finally, there are many cases when a user running the experiment is not its creator; e.g., a researcher repeats another's experiment, teams collaborate on a project, or a testbed is used in education. Such a user has a particularly hard time to detect and diagnose unexpected experiment behavior, and would benefit from automated health management.

In this paper we introduce a novel notion of *experiment health* in Section 2, which denotes the fact that experiment behavior matches some critical set of its creator's expectations. We propose an automated health management infrastructure in Section 3 and discuss its benefits both to users and to testbeds. The infrastructure consists of an expectation language through which a user expresses her expectations about experimental behavior, monitoring agents that collect statistics of experiment behavior, health evaluators that process statistics and compare them to expectations to detect failures, recovery engines that restore experiment health, and a shared library of observations and health tools. We survey related work in section 4 and conclude in Section 5.

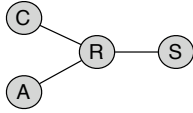


Figure 1: Topology for the SYN Flood experiment.

2. EXPERIMENT HEALTH

We start by providing a sample security experiment that we will use throughout the paper to illustrate our points. The experiment is a *SYN flood attack* that occurs on the topology shown in Figure 1. It generates some Web traffic between client C and server S, then generates a TCP SYN flood from the attacker node A to S. A TCP SYN flood attack [1] denies service to new TCP connections on the port it targets by generating a lot of fake connection records and tying up the server’s OS memory. The denial can be noticed by observing C’s traffic that becomes populated by repeated SYN packets because C cannot establish new connections.

We define experiment health as the property of the experiment behaving according to its creator’s expectations, at all observation levels. To see why health at all levels is important imagine the case of a misconfigured SYN Flood experiment. SYN traffic is sent to another port than 80, which should not deny service to C’s Web traffic. The user can easily spot that something is wrong by observing lack of SYN repetitions in C’s traffic, although she may have trouble diagnosing the cause. Now imagine that in addition to the previous misconfiguration the machine R freezes and stops forwarding traffic. The effect the user expects to see – C repeating SYN packets – will occur, but for the wrong reason. This is why experiment health encompasses experiment behavior at all levels and not just the end outcome. C’s service should be denied because of the SYN flood and not for any other reason. Even this simple example brings home the point that defining experiment health is a complex task. We will describe in Section 3 how we plan to ease it for the user.

It may seem that maintaining experiment health is very similar to network management, whose goal is to support the network mission of delivering traffic effectively between end nodes by maintaining connectivity, distributing load, ensuring correct network configuration, etc. The main difference is that the definition of experiment health depends critically on experimenter’s research goals. Unlike network management that maintains common, well-understood network performance targets, experiment health management must understand what a user wants and maintain that specific behavior. In the example of SYN flood attack, the goal was to deny service – quite opposite from the network management’s goal of keeping networks up and running.

2.1 Failure Causes

We now briefly discuss why an experiment may fail to meet user’s expectations. An experiment’s behavior is influ-

enced by the following internal and external elements, which we believe form an exhaustive list: (1) *Experiment definition and configuration*, including the software and scripts written by the user, topology settings, location and resources of the experimental hardware, and default OS and applications loaded by the testbed on the experimental hardware, (2) *Testbed infrastructure*, such as experimental node hardware, switching fabric, control network, and shared testbed services like DNS and network file system, (3) *Resource sharing* with other experiments, and (4) *The public Internet*, if any of the experimental traffic is expected to cross it. Any of these four elements may behave in a way unforeseen by a user, and interfere with experimental results.

An experiment may be misconfigured, either by inclusion or by omission. Misconfiguration by inclusion means that some feature is explicitly present in the experiment’s setup but it is set to a wrong value, e.g., a traffic generator is engaged at a rate that is too low or a topology is disconnected. We had such misconfiguration in the above example with SYN packets going to a wrong port. Misconfiguration by omission means that a user implicitly assumes presence of some feature/value pair but the assumption does not always hold, e.g., a user assumes that an experimental application always works correctly, but its behavior is actually hardware-dependent, thus the experiment fails when assigned some hardware types.

Any of the critical pieces of testbed infrastructure may fail, including shared services. It is particularly difficult for users to detect or anticipate this kind of failure, since many testbed internals are usually hidden from them.

Resource sharing may create interference between experiments. An experiment may overload the shared resource, such as a file system, if fair sharing is not strictly enforced. An experiment may also interact with a shared application improperly causing it to misbehave and influence other users. In the DETER testbed we have observed experiments crashing the shared server that controls experimental events by sending it overly long messages. Again, users have a hard time diagnosing misbehavior that stems from resource sharing since sharing is implicit in the testbed’s design, whose details are not publicized to users.

Finally, if any experimental traffic travels over the Internet, connectivity, congestion and any Internet services accessed by it may influence experimental behavior. For example, if SYN Flood experiment were to be performed on Planetlab [2] there may be connectivity between R and S but excess SYN traffic may be filtered by some ISP-level defense on this link. This may or may not lead to denial of service for C, depending on the exact filtering behavior.

3. HEALTH MANAGEMENT INFRASTRUCTURE

We propose a health management infrastructure that can be integrated with current testbeds to improve their usability and robustness. We plan to integrate this infrastructure with

```

operations: operations operation
| operation

operation: action expect cond IFFAIL recovery

expect: EXPECT
| EXPECT ALWAYS
| EXPECT ALWAYS CHECK NUMBER
| EXPECT AFTER NUMBER

action: /* empty */
| STRING

recovery: STOP
| LCURLY statements RCURLY

statements: statements statement
| statement

statement: action EXPECT condition IFFAIL recovery
| action
| STOP

cond: /* empty */
| STRING
| LPAREN rexpression RPAREN
| NOT cond
| LPAREN cond RPAREN
| cond AND cond
| cond OR cond

rexpression: aexpression EQ aexpression
| aexpression NE aexpression
| aexpression LT aexpression
| aexpression GT aexpression
| aexpression LE aexpression
| aexpression GE aexpression

aexpression: STRING
| NUMBER

```

Figure 3: Expectation language grammar.

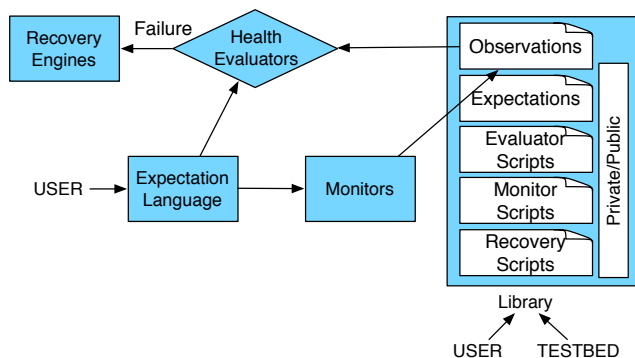


Figure 2: Health Management Infrastructure.

the DETER testbed [3]. Figure 2 shows key pieces of this infrastructure and how they relate to each other. Users express their expectations using the expectation language. The expectations then drive setup of monitors that collect statistics and store them in the library. Periodically health evaluators read the expectations and evaluate statistics against them. On mismatch, failed health is detected and an evaluator triggers recovery engines to restore the health or inform the user. Specifics of desired recovery actions are also obtained from expectations. In addition to observations, the library may hold expectations, together with custom monitor, evaluator and recovery scripts that a user wants to reuse. Each library component can be marked as private or public, thus users may share their data and scripts with others. Testbed operators populate the public library with commonly used scripts and expectations such as the expectation of all experimental machines being up and accepting logins, and all experimental links being passable.

With regard to deployment, we envision some monitors running on experimental nodes (such as those monitoring file system changes) and others running on the testbed infrastructure (those monitoring the testbed’s health). Placement of health evaluators and the library must be such to minimize communication cost for library reads and writes, while maximizing sharing and robustness. In a geographically centralized testbed like Emulab [4] is likely that both

these services would run on shared testbed nodes. In a distributed testbed like Planetlab [2] creating a local copy of relevant library modules and deploying evaluators on a dedicated experimental node may yield better performance. Recovery engines are instantiated on need bases, and their location depends on the type of recovery.

3.1 Expectation Language

Figure 3 shows the preliminary grammar of the expectation language in yacc format. We expect to extend this design as our work progresses.

Expectation statements start by specifying some action, such as calling a user-supplied script. After the keyword EXPECT, the user specifies what the desired outcome should be by invoking health evaluators (see next section). If the expectation is not met, actions after the IFFAIL keyword are invoked. Within the IFFAIL body the user may specify additional expectations, invoke recovery actions (such as sending herself an email), or stop the execution using the keyword STOP. Commands that specify health evaluators and recovery actions refer to corresponding library scripts. An EXPECT keyword can be followed by ALWAYS denoting that health evaluation should be repeated periodically. The default period of 1 minute can be changed using the keyword CHECK followed by a number of seconds. An EXPECT AFTER followed by a number of seconds denotes evaluation that should be performed after that delay, to let the action take effect.

We now illustrate how a user may use expectations to detect and diagnose failures. While early experimentation requires user interaction with the experiment, mature experimentation is often scripted so many runs can be executed to collect statistically significant data. Expectations are meant to extend existing scripts with failure detection and diagnostics. Figure 4 shows a sample expectation script for the SYN flood attack. Text shown in black is the original user’s script that starts the legitimate traffic and starts the attack. The first line invokes library functions that continuously check if all nodes and links are functioning. The second line checks for presence of SYN cookies [5], a popular defense against SYN flood that is on by default in some operating systems.

```

EXPECT ALWAYS all_nodes_up AND all_links_up
  IFFAIL { alert("Testbed hardware problem")
  STOP }
EXPECT [NOT exists("/proc/sys/net/ipv4/tcp_syncookies", S)
OR contents("/proc/sys/net/ipv4/tcp_syncookies", S, "0")]
  IFFAIL { alert("Syncookies are on at S")
  STOP }
run_leg_traffic EXPECT traffic(C,S,"dst port 80")
  IFFAIL { alert("Legitimate traffic does not flow")
  STOP }
run_attack EXPECT traffic(A,S,"dst port 80 and 'tcp[13] = 2'")
  IFFAIL { alert("Attack does not occur")
  STOP }
EXPECT AFTER 10 [traffic(C,S,"dst port 80 and 'tcp[13] = 2'")
> traffic(C,S,"dst port 80 and 'tcp[13] & 8 == 8'")]
  IFFAIL { alert("Attack does not work")
  STOP }

```

Figure 4: Expectations for SYN flood attack.

Legitimate traffic is then run and an evaluator checks if it arrives successfully at the server; the same check is repeated for attack traffic in the next line. Finally, the attack’s success is checked by verifying that the number of SYN packets is larger than number of data packets arriving at the server. On all failures the user is alerted via a message that helps diagnose the source of failure, and execution is halted.

Expectations may be written by an experimenter, or a testbed may attempt to aid expectation writing by providing support for user action observation, correlation and learning. We plan to first implement explicit expectations in DETER and then add the following learning approach. The testbed monitors the user’s shell history and correlates this with traffic, file and process changes observed on all experimental machines, at a relatively low frequency, customizable by the user. Changes that occur in a small window after a user’s action are recorded as an expected outcome. It is likely that several iterations will be needed to discern cause-effect changes vs changes that just happen to overlap a user’s action but are not causally related. We also anticipate that inferred expectations will be incomplete and imperfect — an automated process can hardly match a human’s understanding of experimental behavior. We emphasize however that even human-designed expectations are likely to evolve over a life of an experiment as the user discovers new behavior invariants and failure modes. The learning approach we propose is simply meant to provide a rough starting point that must further be iteratively refined by the user.

3.2 Monitors

To support health evaluation it is critical to provide monitoring and data collection about what is happening in the testbed as a whole and in each particular experiment. There are three key sources of such data: (1) static testbed data such as IP addresses, node allocation to a particular experiment, etc. (2) data collected routinely in all experiments and in the facility itself, such as packet traces, node liveness, etc., and (3) explicit data collection requested by an experimenter or the testbed management, in context of an expectation’s condition evaluation. We also identify three challenges to data collection. First, because of scale and system unreliability, available data may be incomplete. Second,

the desired information may not be directly measurable, but must be inferred from other measurements that can be gathered (e.g. quality of service). Finally, in light of security expectations that relate to privacy, some information may not be accessible where it is desired. Either an experiment or the testbed system may withhold information from the other. We will provide a base set of data collection tools in our library, which will be extensible by researchers. Specifically we will provide tools that collect traffic, process status data, file system data and file contents.

There is also an inherent tradeoff in data collection that influences its frequency. Too frequent collection may interfere with the experiment’s behavior, but infrequent collection may delay detection of failures. It will likely be impossible to set a default frequency that satisfies all users, since each experiment’s sensitivity to data collection depends on user-specified experiment’s behavior, and users may be satisfied with a wide range of detection delays. Instead we hope to provide a safe, low-frequency setting as a default for novice users and a monitor API for advanced users to customize frequencies. We acknowledge that getting these settings right will be challenging for users but we do not see a better alternative.

3.3 Health Evaluators

The job of health evaluation is to determine whether the current behavior of a subject meets specified health criteria. There are two aspects to evaluating the health of a subject. The first is to select the particular behaviors of the subject, such as link bandwidth, jitter or loss rate, that are to be evaluated for the desired expectation goal. The second is to determine if the health of that subject meets the desired goal, i.e. to translate a quantitative result such as a link’s loss rate into a qualitative true/false result. We plan to implement evaluators that must be explicitly invoked by users, with success criteria that must also be explicitly specified. Initial evaluators provided in the library will help users detect if a machine is live, if there is a shell on that machine, if a link is passable, if a file exists, if file contents match a regular expression supplied by the user, if a process exists and matches a specified state, and if specific traffic flows between two nodes.

Returning to Figure 4, evaluators `all_nodes_up` and `all_links_up` are complex library functions that call simple functions `node_up(node)` and `link_up(link)` for each node and link in the experiment and return true if all evaluations are true. Evaluator `exists` checks for presence of a file on a specific experimental node — in our case node S. Evaluator `contents` is similar to `exists` but has a third argument — a regular expression that should match file contents for the result to be true. Evaluator `traffic` checks if traffic flows from the first node in the argument list to the second, and returns packet count within some prespecified interval. The third argument is a `tcpdump` expression that is used to filter traffic of interest.

We now touch on the problem of composite evaluation

```

run_attack EXPECT traffic(A,S,"dst port 80 and 'tcp[13] = 2")
  IFFAIL {
    EXPECT exists("run_attack",A) IFFAIL
      { alert("Attack file does not exist")
        STOP }
    alert("Attack script does not run")
    STOP }}

```

Figure 5: Recovery actions based on failure cause.

in the case of more complex goals. For example, our goal may be to create the SYN flood attack from 20 machines to a server, then engage some defense on several routers in the topology that are on the traffic’s path, and measure if it is working. Expressing this goal via low-level expectations would result in a long, unwieldy script. We will likely need to extend the expectation language vocabulary in the future for iteration, expectation parameterization and composition of high-level expectations by invoking low-level ones by name, similar to function calls.

3.4 Repair

Repair actions are invoked on a detected failure. The action may be enclosed into an expectation statement if there are different recovery steps that may be taken depending on problem diagnosis. For example, imagine that the `run_attack` script may fail either because there is no such file or because the commands inside the script do not run. The fourth line from Figure 4 is modified as shown in Figure 5 to account for these two sources of error, diagnose them properly and take correct recovery action.

Similarly to monitoring tools, we will initially develop several common recovery tools and store them into the public library: emailing the user or the operator, displaying a pop-up window to the user, rebooting a machine, swapping out the experiment, reswapping the experiment, quarantining the experiment, reconfiguring the experimental firewall or the main DETER firewall, and stopping, starting or restarting a process.

3.5 Sharing

We expect that a lot of tools developed for experiment health management will be reusable. Tool sharing occurs via the library that we will initially populate with public, monitoring and recovery scripts, and health evaluators. The library will be extensible, so that users can reuse their custom scripts and can build upon each other’s work. Users will be able to mark library components they contribute as either public or private.

The collected information may be valuable to more than one monitoring tool in different contexts. For example, it may be important to collect packet traces in an experiment for the following uses: (1) user’s evaluators may depend on these traces to verify correct experiment behavior, (2) the testbed may use such traffic information to determine the health of the complete set of resources it is managing, (3) testbed operators may use it to develop long-term traf-

fic and usage models for planning provisioning, (4) traffic traces may provide an audit trail if an experiment creates a security risk for the testbed. The information should be collected once and managed effectively to allow for multiple uses. This requires a common information substrate or information plane, with well defined access rules and contexts, that we plan to develop on DETER. Information collection and access must be designed to reflect the security and privacy expectations both of users and of the testbed.

3.6 Benefits of Health Management

We now summarize the expected benefits from experiment health management. While many users and all testbed operators already deploy custom scripts to monitor the experimental and testbed behavior, there are several drawbacks to this practice. First, scripts are usually written based on prior experience with common failure modes and an implicit definition of desired behavior. They are often engaged on need basis. The health management framework encourages explicit specification of desired behavior and engagement dynamics thus helping users and operators become aware of, formalize and refine their assumptions. Second, scripts are often tied to a single experiment or testbed, and hard to share or reuse while we hope to develop parameterized monitors, health evaluators and recovery engines that facilitate sharing and can be reused without modifications.

4. RELATED WORK

Our experiment health management is an extension and customization of knowledge plane [6] ideas to network testbeds. Much recent work in this area has concentrated on the sub-problem of supporting an information plane [7, 8, 9, 10] but it does not provide the ability to control or limit access to information based on security and privacy policies, which we plan to develop.

There is vast work in network management, which differs from our health management because testbed experiments lack generally agreed-upon correctness or performance goals. Another difference lies in the granularity at which management is done: networks are managed at the high granularity of network elements and links, while experiments also need to be managed at the low granularity of user actions.

There are a number of tools for distributed application management on PlanetLab [2], of which the most advanced is Plush [11]. Plush is a toolkit for distributed application configuration, management and visualization. Plush enables users to specify tasks in XML format, then executes them invoking low-level process, file and resource monitoring to detect failures. Plush also provides synchronization primitives and performs resource acquisition and reallocation as needed. The primary distinction between Plush and our proposed health monitoring is that Plush manages for known performance goals (connectivity, process liveness, etc.) suitable for continuously running applications, while we addi-

tionally manage for customizable performance goals that are suitable for widely varying testbed experiments. Our management thus includes the notion of “expected performance” and covers a wider range of behaviors than Plush.

Emulab’s Experimenters Workbench [12] contains support for experiment versioning, cloning via a template, and archiving. These capabilities support pre-packaged experiments, and are needed for sharing but they do not provide any support for experiment creation and correctness checking, which is our main focus.

In the area of expectation or policy specification languages, we mention two extremes. XACML [13] is a declarative language and expects something else to enforce policies. From our perspective, policy declarations are only a small part of our challenge. In contrast, Ponder [14, 15] is an object-based language for declaring not only security and management policies, but time, state, and composite conditions under which they should be evaluated, sets of subjects to be evaluated, sets of targets over which some action might be taken, and the actions themselves. Simpler than Ponder, Tcl Expect [16] is a scripting environment whose syntax enables specification of control flows that depend on controlled program outputs, thus automating system testing. Our expectation language will incorporate capabilities present in Ponder and Tcl Expect, but our challenge is to make the language easily accessible, usable, and understandable by a broad set of differently skilled experimenters. We thus plan to reuse features of existing languages, wrapping them in more user-friendly syntax and higher level language constructs.

5. CONCLUSIONS

We have described an *experiment health management* system for use by cybersecurity researchers conducting experimental testbed research. The system allows researchers to explicitly express invariant conditions or *expectations* about an experiment’s behavior. It monitors these expectations during experiment execution, and may trigger reporting, logging, or corrective actions if expectations are not met.

We see two distinct advantages to the use of such a system. First, the very act of making expectations explicit improves research methodology and leads to improved understanding. The act of documenting experiment invariants within our system may help the researcher to carefully consider the scope and validity of their experiment. Once complete, this documentation will communicate the experimenter’s assumptions and expectations to other researchers and students, promoting repeatable research.

The system’s second advantage is its ability to improve the validity of experimental results, by detecting failures of experiment configuration, setup, or underlying infrastructure that might otherwise go undetected. This capability increases quickly in importance as experiments grow in scale and complexity.

The system described in this paper represents a first step towards a comprehensive and usable experiment health monitoring system. To fully realize the potential of such a system, further advances will be required both in functionality of the system and ease of use. High level, intuitive tools that assist the experimenter in capturing expectations and developing appropriate responses to failures will be particularly important. Nonetheless, even this first step may be of significant value - to researchers and teachers willing to take the time to draft expectation scripts; to students who find the resulting expectation documentation helpful in understanding the intent of an experiment, and to those seeking to make further progress in experimental cybersecurity research methodology itself.

6. REFERENCES

- [1] Wikipedia. SYN attack. http://en.wikipedia.org/wiki/SYN_flood.
- [2] L. Peterson, A. Bavier, M. Fluczynski, and S. Muir. Experiences Building PlanetLab. In *Proc. of Operating Systems Design and Implementation Symposium (OSDI)*, November 2006.
- [3] Terry Benzel, Robert Braden, Dongho Kim, Clifford Neuman, Anthony Joseph, Keith Sklower, Ron Ostrenga, and Stephen Schwab. Experience with DETER: A Testbed for Security Research. In *Proc. of Tridentcom*, March 2006.
- [4] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, December 2002.
- [5] D. J. Bernstein. SYN Cookies. <http://cr.yip.to/syncookies.html>.
- [6] D. D. Clark, C. Partridge, J. C. Ramming, and J. T. Wroclawski. A Knowledge Plane for the Internet. In *Proc. of ACM SIGCOMM*, 2003.
- [7] M. Wawrzoniak, L. Peterson, and T. Roscoe. Sophia: An Information Plane for Networked Systems. In *Proc. of 2nd Workshop on Hot Topics in Networks*, pages 15–20, 2004.
- [8] J. Kopena and B. Loo. OntoNet: Scalable Knowledge-Based Networking. In *Proc. of 4th Workshop on Networking Meets Databases*, 2008.
- [9] Y. Mao, B. Loo, Z. Ives, and J. Smith. The Case for a Unified Extensible Data-Centric Mobility Infrastructure. In *Proc. of ACM SIGCOMM International Workshop on Mobility in the Evolving Internet Architecture*, 2007.
- [10] W. Zhou, E. Cronin, and B. Loo. Provenance-aware Secure Networks. In *Proc. of 4th Workshop on Networking Meets Databases*, 2008.
- [11] J. Albrecht, C. Tuttle, A. C. Snoeren, and A. Vahdat. PlanetLab Application Management Using Plush. *ACM Operating Systems Review (SIGOPS-OSR)*, 40(1), January 2006.
- [12] E. Eide, L. Stoller, and J. Lepreau. An Experimentation Workbench for Replayable Networking Research. In *Proc. of Network Systems Design and Implementation Symposium (NSDI)*, 2007.
- [13] OASIS. OASIS extensible Access Control Markup Language. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml.
- [14] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder Policy Specification Language. In *Proc. of Workshop on Policies for Distributed Systems and Networks*, 2001.
- [15] N. Dulay, E. Lupu, M. Sloman, and N. Damianou. A Policy Deployment Model for the Ponder Language. In *Proc. of IEEE International Symposium on Integrated Network Management*, 2001.
- [16] D. Libes. expect: Curing those Uncontrollable Fits of Interaction. In *Proc. of the Summer USENIX Conference, Anaheim*, June 1990.