Automatic Proof of Survivability Compliance –Approaches and Techniques

Yanjun Zuo

***Abstract*—Survivability represents a system's capability to withstand malicious attacks and system failures in order to provide essential services to users even in a challenging environment. In a proof-carrying paradigm, a user publishes his/her survivability requirement policy and a system provider constructs a proof that the system satisfies the user's requirements. Finally, the user verifies if the proof is valid. In this paper, we discuss proof approaches and techniques used by the system provider to automatically compile such a proof. We develop algorithms to show how different proof choices are generated so that the system provider can choose the most cost-efficient approach in the proof process. Proof generation relies on the certifications generated by trusted evaluators. We show the necessary steps to construct the basic proof elements which can be logically linked to form the ultimate proof.**

***Index Terms*—Algorithm, proof, survivability, verification.**

I. Introduction

Our society is increasingly dependent on large-scale, interconnected systems. Any system, which provides vital services to the nation and the society, must be reliable and dependable. Survivability is defined as the ability of a system to provide essential services in the presence of attacks and failures, and recover full services in a timely manner [1]. Survivability has been considered as a key inherent property of a reliable system [2].

Given the critical nature of survivability in many high-security and high-integrity settings, there is a need to develop formal evaluation and verification models to systemically prove that the system has the required survivability features. In this paper, we propose such a formal approach for survivability proof by a system provider. Our focus is on the generation of a set of proof choices for the system provider to choose and determination of a set of steps to compile a valid proof. A proof represents formal evidence that the system under evaluation satisfies the user's survivability requirements. The proof can be verified by a trustworthy checker program. Since our approach can facilitate automatic proof generation and verification, it is possible for the user to assess the survivability features of a system real time and to accept the system only if it meets the user's requirements. If so, all the survivability requirements as set by the user are guaranteed to be satisfied. Any system that does not meet those requirements will be detected before the system is deployed.

By shifting the proof burden from the system user to the system provider, the latter can better use their knowledge about the features of their own system in constructing a compliance proof. The system provider is supposed to know better than anyone else why their system satisfies the user's survivability requirements, i.e., which survivability properties their system has as required by the user. The system user only needs to define their particular survivability requirements for the system of interest and to verify the proof once it is submitted by the system provider to confirm that the system satisfies those requirements.

Manuscript received June 15, 2012; revised August 5, 2012.

Yanjun Zuo is with the University of North Dakota, Grand Forks, ND, USA 58201.

II. Related Work

The most related research to our study is proof-carrying code (PCC) and authentication. PCC is a software mechanism which constructs and verifies a mathematical proof about the machine-language program and guarantees its safety [3]. In most approaches to PCC (e.g., [3-4]), the machine-checkable proofs are written in a logic with a built-in understanding of a particular type system. Such a PCC system must understand the language of types and the machine language for a particular machine. Those proof-carrying codes are highly type-specialized, and they mainly address the issue of programming language safety. There are other practical applications of PCC. For instance, PCC was used to implement a collection of network packet filters [5] and applied to access-control, distributed authorization, and policy-specification language (e.g., [6]-[9]).

Our research can be used as complementary techniques in proof-carrying diagrams. The main contribution is the integration of survivability techniques and reasoning with a proof-carrying framework. We discuss the approaches and techniques to prove that a system satisfies the user's survivability requirements. An algorithm is developed to show the proof choices available to a system provider who can choose the most cost efficient one to construct a proof. We also show the step-by-step procedure to generate a proof using the specified survivability specific operators.

III. System Architecture and Example of Survivability Requirement Policy

Fig. 1 shows the system architecture and the major components of the framework. The system provider is an entity to supply the software system. The system user is the consumer of the system. It accepts the system only if the system provider can prove that the system satisfies its survivability requirements. In high-security and high-integrity applications, any external software object to be

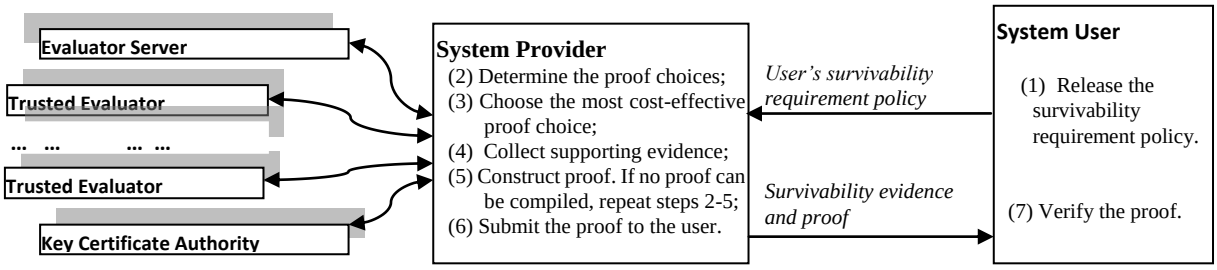

Fig. 1. The system architecture.

acquired must satisfy a user's survivability requirements. Those requirements are specified as the user's policy. To show their system's compliance to the policy, the system provider needs to compile and submit a proof. Proof generation relies on the certifications generated by trusted evaluators. The system provider first collects evidence from the trusted evaluators who can confirm that the system has the required survivability features and then applies the appropriate approaches and techniques to construct the proof. Finally, the system user verifies whether the proof is valid. If so, the system can be considered as satisfactory and acceptable. To better illustrate our approach, we use a hypothetical command and control system cited in [10] as a running example throughout the paper. The user's survivability requirements are represented in a tree structures, called a *survivability requirement tree* (Fig. 2).

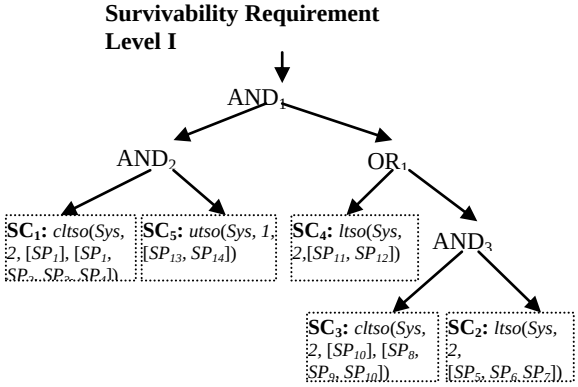

Fig. 2. The example of survivability requirement policy.

The survivability requirements are specified from four dimensions (called survivability characteristics) as shown in Table I. Each dimension contains multiple refined survivability properties called survivability primitives. A survivability characteristic can represent a desired or unwanted system feature. For a given desirable survivability characteristic *SC*, the low bound threshold operator *ltso(Sys, i, [SP$_1$, SP$_2$, …, SP$_n$])* indicates that a system *Sys* must have contributions for at least *i* out of *n* survivability primitives to be considered as satisfying the requirements of *SC*. Furthermore, the conditional low bound threshold operator *cltso(Sys, i, [SP$_j$,…, SP$_k$],[SP$_1$, SP$_2$, …, SP$_n$])* indicates that a system must have

contributions for at least *i* out of *n* survivability primitives and *SP$_j$, …, SP$_k$* must be satisfied.

For a undesirable survivability characteristic *SC'*, the *upper-bound threshold selection operator*, denoted as *utso(Sys, j, [SP$_1$, SP$_2$, …, SP$_m$])*, indicates that the system must not have concerns for more than *j* out of the *m* survivability primitives. If so, the system will be considered not going beyond a user's concerns from the perspective of *SC'*. Essentially, such an operator defines the "most tolerable" bound for potential unfavorable features of a system from the perspective of a survivability characteristic.

TABLE I: SURVIVABILITY CHARACTERISTICS AND PRIMITIVES

Survivability Characteristics	**Survivability Primitives**
Adaptability (SC$_1$)	Monitoring and control (SP$_1$) Self-reconfiguration (SP$_2$) Process migration (SP$_3$) Service prioritization (SP$_4$)
Recoverability (SC$_2$)	System self-healing (SP$_5$) System roll-back (SP$_6$) Malice immunization (SP$_7$)
Fault tolerance (SC$_3$)	Redundancy and diversity based fault masking (SP$_8$) Fault isolation and system partition (SP$_9$) Tolerance through backup (SP$_{10}$)
Reliability (SC$_4$)	Service availability (SP$_{11}$) Service consistency (SP$_{12}$)
Performance Degrading (SC$_5$)	Communication degrading (SP$_{13}$) Operation degrading (SP$_{14}$)

Following the above variable and operator definitions, the user's survivability requirement policy as shown in Fig. 2 indicates that the system must satisfy the following three conditions in order for the system *Sys* to be considered at survivability level I:

1) at least two survivability primitives as defined for system adaptability (*SC$_1$*) and *SP$_1$* must be satisfied;
2) either *condition 1* (at least two survivability primitives defined for system reliability (*SC$_4$*)) or *condition 2* (at least two survivability primitives defined for system fault tolerance (*SC$_3$*) and system recoverability (*SC$_2$*) must be satisfied. In addition, one of the two primitives for *SC$_3$* must be *SP$_{10}$*); and
3) no more than one concern for the survivability primitives in terms of system performance degrading.

In a survivability requirement tree, the root represents the desired survivability level. Each leaf node represents the requirements in terms of a survivability characteristic (represented in *ltso*, *cltso*, or *utso*). An intermediate node represents a logical operator (*AND*, *OR*). To distinguish different logical operators, we add a subscript to each of them without changing their meanings.

IV. DETERMING PROOF OPITONS

An important part of the proof process is for the system provider to decide a proof choice to construct a proof that its system satisfies the user's requirements. A proof choice indicates which set of requirement elements (e.g., survivability characteristics) to prove based on the user's

requirements. Intuitively, a proof choice is to determine which branch of each *OR* node to choose in the survivability requirement tree. For instance, since the user's survivability requirements as shown in Fig. 2 has only one *OR* node, the system provider has two choices (see Fig. 3) to choose to prove that the system satisfies the survivability level I: (1) to prove (pf_1 and pf_2) and pf_3; or (2) to prove (pf_1 and pf_2) and (pf_4 and pf_5). We call such a pf_i a *proof construct*. Each proof construct corresponds to a survivability characteristic represented by *ltso*, *cltso*, or *utso*.

$Pf_1 \equiv$ Adaptability (SC_1): *cltso*(*Sys, 2,* [SP_1], [SP_1, SP_2, SP_3, SP_4])
$pf_2 \equiv$ Acceptable Performance (SC_5): *utso*(*Sys, 1,* [SP_{13},SP_{14}])
$pf_3 \equiv$ Reliability (SC_4): *ltso*(*Sys, 2,*[SP_{11}, SP_{12}])
$pf_4 \equiv$ Fault Tolerance(SC_3): *cltso*(*Sys, 2,* [SP_{10}], [SP_8, SP_9, SP_{10}])
$pf_5 \equiv$ Recoverability(SC_2): *ltso*(*Sys, 2,*[SP_5, SP_6, SP_7])

***Proof Choice One*:** to prove $(pf_1\ AND_2\ pf_2)\ AND_1\ pf_3$
Proof Choice Two: to prove $(pf_1\ AND_2\ pf_2)\ AND_1\ (pf_4\ AND_3\ pf_5)$

Fig. 3. Survivability proof choices.

A. Determining the Number of Proof Choices

We show how to calculate the number of proof choices given a survivability requirement tree *T*. It can be observed that the number of the choices is mainly determined by the number of *OR* nodes in *T*. We first define a variable associated with each node *N* (except the root), denoted as *count*(*N*), which represents the number of proof choices to prove this node. If *N* is a leaf node, then *count*(*N*) =1. For any intermediate node *N*, *count*(*N*) is calculated as a function of the count values of its child node and the type of *N*. More specifically, let's consider a node *N* with *m* children node $N_1, \ldots, N_m$. There are two cases:

Case 1: N is an *AND* node. Then $count(N) = \prod_{i=1}^{i=n} count(N_i)$;

Case 2: N is an *OR* node. Then $count(N) = \sum_{i=1}^{i=n} count(N_i)$.

We have developed an algorithm (see Algorithm One) to show the calculation process. For an input survivability requirement tree *T*, the algorithm first assigns each leaf node with a count value 1. Then, it recursively calculates the count value of a parent node from the count values of its children nodes in a bottom-up fashion until the root node is reached. Since all the nodes have to be processed, the time complexity of the algorithm is O(*n*), where *n* represents the number of nodes in *T*.

Algorithm One: Proof Choices Counting main *PChoice_Counting(T)*

4) For each node *N* in *T*, set *count(N)* = 1;
5) Call *Node_Processiong1*(N_0) with the child node N_0 of the root, which returns the number of proof choices for *T*.

subroutine *Node_Processing1 (N)*

6) Let $L=\{N_1, N_2, \ldots, N_m\}$ represent the children nodes of *N ;*
7) If *N* has no more child node, return *count*(*N*);
8) Else, do the following:
   - For each of the child node N_i ($1 \le i \le m$), do: *count*(N_i) = *Node_Processing1*(N_i);
   - If *N* is *AND* node, return $count(N) = \prod_{i=1}^{i=n} count(N_i)$;
   - Else, return $count(N) = \sum_{i=1}^{i=n} count(N_i)$;

B. Identifying the Proof Choices

Determining an efficient proof choice is critical in order to generate the most efficient proof. A choice determines which set of evidence to collect. As we discussed earlier, a piece of evidence refers to an evaluation certificate issued by a trusted evaluator which confirms that the system satisfies the user's requirement in terms of a particular survivability primitive. Each proof is associated with a cost in terms of such factors as the time for the evaluator to inspect the system and issue a compliance certificate. The system provider always chooses the most cost efficient proof choices to pursue from all choices. After a choice is identified, the necessary survivability primitives and characteristics to be proved can be determined. Algorithm Two show how the proof choices are generated given a survivability requirement tree *T*. The system provider can choose the proof choice with the lowest cost to compile a complete proof and then submit it to the system user.

Algorithm Two: Proof Choice Generation

Input: survivability requirement tree *T*

Output: A set of proof choices $S = [pf(N_1) \wedge \ldots \wedge pf(N_k)], \ldots [pf(N'_1) \wedge \ldots \wedge pf(N'_k)]$.

main *PChoice_Generation(T)*

1) Let N_0 represent the child node of the root of *T*;
2) Call the sub routine *Node_Processing2*(N_0), which returns the set of proof choices, i.e., *S* = *Node_Processing2*(N_0).

subroutine *Node_Processing2(N)*

1) Let $L=\{N_1, N_2, \ldots, N_m\}$ represent the children nodes of *N*.
2) Let set *S* represent the partial results containing the proof choices. Initially, $S = \varepsilon$.
3) If *N* has no more child node, return pf(*N*); *//N is a leaf node*
4) Else, for the first child node N_1, do the following:
   - *S= Node_Processing2*(N_1)*.return Set*;

We write $S' = A_1 \vee \ldots \vee A_k$ where each A_i is in a sum-of-products canonical form $A_i = a_1 \wedge \ldots \wedge a_n$.

- For each of the child node N_i ($2 \le i \le m$), do:

S' = Node_Processing2(N_i)*.return Set*;

We write $S' = B_1 \vee \ldots \vee B_t$ where each B_j is in a sum-of-products canonical form.

If the parent node *N* is an *AND* node, do

$S = (A_1 \wedge B_1) \vee \ldots \vee (A_1 \wedge B_t) \vee \ldots \vee (A_k \wedge B_1) \vee \ldots \vee (A_k \wedge B_t)$;

Else, do // *N is an OR node*

$S = A_1 \vee \ldots \vee A_k \vee B_1 \vee \ldots \vee B_t$;

5) returnSet *S*. // returns the set *S* to the calling function

The survivability requirement tree *T* is processed starting from the root. We use the term pf*(N)* to represent the proof of a node *N* of *T*, where *N* can be a logical operator node (i.e., *AND*, *OR*) or a threshold selection node (*ltso*, *cltso*, or *utso*). If *N* represents a logical node, pf*(N)* is converted to the proofs of the children nodes of *N*, i.e., pf(N_1), …, pf(N_m), where $N_1, \ldots, N_m$ represent the children of *N* as represented in *T*. For instance, pf(*AND*)= pf$(N_1) \wedge \ldots \wedge$ pf(N_m) and pf(*OR*)= pf(N_1)

$\vee \ldots \vee pf(N_m)$. As we can see, to enumerate all the proof choices, we recursively replace the proof of each parent node with the proofs of its children nodes until all the children nodes represent the threshold selection operators. However, this recursive processing must maintain the logic Boolean sum-of-products (SoP) canonical form $[pf(N_1) \wedge \ldots \wedge pf(N_k)] \vee \ldots [pf(N'_1) \wedge \ldots \wedge pf(N'_k)]$. If necessary, the distribution rule $[pf(N_i) \vee pf(N_j)] \wedge pf(N_t) = [pf(N_i) \wedge pf(N_t)] \vee [pf(N_j) \wedge pf(N_t)]$ must be applied. In this way, the algorithm guarantees that each child node only returns the proof terms in a SoP canonical form when called recursively while its parent node is being processed. For each parent node, the algorithm processes the sub proofs from its children nodes from left to right, fusing the partial result with the proof of the next child node following the two cases below:

1) if the parent node is an *AND* node, then we have the format $(A_1 \vee \ldots \vee A_k) \wedge (B_1 \vee \ldots \vee B_t)$, where each A_i represents the partial proof terms in a maxterm canonical form; and each B_j represents a conjunction proof term from a child node. Then $(A_1 \vee \ldots \vee A_k) \wedge (B_1 \vee \ldots \vee B_t)$ can be normalized to $(A_1 \wedge B_1) \vee \ldots \vee (A_1 \wedge B_t) \vee \ldots \vee (A_k \wedge B_1) \vee \ldots \vee (A_k \wedge B_t)$, which is in a SoP canonical form;
2) if the parent node is an *OR* node, then we have the format $(A_1 \vee \ldots \vee A_k) \vee (B_1 \vee \ldots \vee B_t)$, which can be easily normalized to $A_1 \vee \ldots \vee A_k \vee B_1 \vee \ldots \vee B_t$ in a SoP canonical form.

The algorithm conducts a depth first search and terminates when all the basic proof terms represent proof constructs. Then, each conjunction term $[pf(N_1) \wedge \ldots \wedge pf(N_k)]$ represents a proof choice, where $pf(N_i)$ represents the proof of a survivability characteristic in terms of a threshold operator (*ltso*, *cltso*, or *utso*) and its arguments. Since the step 4.2.2 in Algorithm Two needs to process all the combinations of the minterms from the proof terms of each child node, the time complexity of Algorithm 2 is $O(n^2)$, where n represents the number of nodes in T.

V. Proof Generation

In this section, we discuss how the system provider compiles a valid proof. The process for proof generation has two sets of tasks which can be executed in a concurrent and interleaving fashion. For the first set of tasks, the supporting evidence which may be useful in proof derivation is collected. For the second set of tasks, a prover program attempts to generate a derivation of the goal statement based on the available evidence and other assumptions (e.g., survivability policy elements). If no sufficient evidence is provided, additional facts must be collected and the two sets of tasks are repeated until either a proof is finally generated or a failure is reported to indicate that no proof can be possibly generated (e.g., the system does not possess all the required survivability properties).

The system provider first identifies the available proof choices and then chooses the one with the lowest cost. A proof choice contains a set of proof constructs $\{pf(N_1), \ldots, pf(N_k)\}$. Each $pf(N_i)$ corresponds to the proof of a survivability characteristic *SC*. To compile $pf(N_i)$, we show how to generate a proof for each survivability primitive in *SC*. We first define the following terms in Fig. 4:

sign(S, K_c):= Statement *S* is signed by an entity with public key K_c;
keyBind(K_c,C):= K_c is the public key of entity *C*;
ensure(C, S):= Entity *C* ensures that statement *S* is true;
cerAuth(CA):= Entity *CA* is a trusted key certificate authority;
trustedEav(C):= Entity *C* is a trusted evaluator to assess the survivability features of a system;
surPrim(SP):= *SP* is a recognized survivability primitive;
sat(Sys, SP):= System *Sys* satisfies the survivability requirements in terms of a survivability primitive *SP*.

Fig. 4. Survivability element proof variables.

The general steps to compile a proof for a survivability primitive *SP* are shown below.

1) $sign(CA, (keyBind(K_c, C))) \wedge cerAuth(CA) \rightarrow keyBind(K_c, C)$

If certificate authority CA signs a certificate for key binding between entity C and the cryptographic public key K_c, then it is believed that K_c is C's public key.

2) $sign(S, K_c) \wedge keyBind(K_c, C) \rightarrow endorse(C, S)$

If statement S is signed by entity C with public key K_c, then it is believed that C endorses S.

3) $trustedEva(C) \wedge endorse(C, S) \rightarrow S$

If an entity endorses a statement and the entity is a trusted evaluator, then the statement is considered true.

4) $(S \rightarrow sat(Sys, SP)) \wedge SurPrim(SP) \rightarrow sat(Sys, SP)$

If statement S indicates that system Sys satisfies the survivability requirement in terms of a survivability primitive SP, it is considered true.

Combined together, the above four steps prove that system *Sys* satisfies a user's requirements in terms of survivability primitive *SP* as evaluated by a trusted evaluator *C*:

$PF: [sign(S, K_c) \wedge trustEva(C) \wedge sign(CA, (keyBind(K_c, C))) \wedge cerAuth(CA) \wedge SurPrim(SP) \wedge S \rightarrow sat(Sys, SP)] \rightarrow sat(Sys, SP)$

Formula *PF* is applied to prove one survivability primitive. For illustration purposes, we assume that evaluators confirmed that system *Sys* is self-reconfigurable and capable of mitigating critical services to clean, healthy components to avoid further damage in case of malicious attacks. Hence, $sat(Sys, SP_2)$ and $sat(Sys, SP_3)$ can be derived by following Formula *PF*. According to the meaning of the selection operators, the system provider can prove that *Sys* satisfies the user's survivability requirements in terms of system *recoverability* (SC_2), i.e., $sat(Sys, ltso(Sys, 2, [SP_5, SP_6, SP_7]))$ or $sat(Sys, SC_2)$.

Furthermore, we assume that the following proofs have been available (all the proofs can be obtained by following the above procedures): *sat(Sys, Acceptable Performance Degrading)*, *sat(Sys, Adaptability)*, *sat(Sys, Fault tolerance)*, and *sat(Sys, Recoverability)*. Those terms show that system *Sys* satisfies the survivability characteristics: adaptability, fault tolerance, recoverability, and acceptable level of performance degrading. If the system provider chooses the proof choice two (see Fig. 3), then the rest of proof is shown in Fig. 5.

$$\dfrac{\dfrac{\dfrac{sat(Sys, SC_5) \wedge sat(Sys, SC_1)}{sat(Sys, AND_2)} \qquad \dfrac{sat(Sys, SC_3) \wedge sat(Sys, SC_2)}{sat(Sys, AND_3)}}{sat(Sys, OR_1)}}{sat(Sys, AND_1)}$$

Survivability Level I

Fig. 5. Partial proof tree.

Finally, the system provider submits a complete survivability proof along with each verification certificate to the system user to verify. Since digital signatures and trusted third parties are used, any proof is tamperproof.

Upon receipt of a survivability proof, the system user applies a checker program to verify that the proof is valid. The proof verification process will be conducted in a bottom-up fashion starting from each proof element (i.e., the proof of survivability primitive). Then, the survivability characteristic corresponding to a threshold selection operator such as *ltso*, *cltso* or *utso* can be verified. Finally, the checker program verifies the complete proof relative to the user's policy.

VI. Simulation

We implemented a prototyping system of the proof generation framework. A prover runs at the system provider side, which automatically generates a proof based on the user's survivability requirements. We measured the time for the prover to compile a proof given a survivability policy. We executed the prover program on a computer running on Windows XP Professional. We collected the execution time by generating a proof for the example survivability policy (see Fig. 2). Multiple executions are conducted, and the average time to generate a proof is 1042 milliseconds. Since the proof generation time increases with the number of proof choices, we also show the proof generation times with different number of proof choices (see Fig. 6). In our future work, we plan to apply multi-threading in our implementation. Since all the proof choices under each *OR* operator can be independently proved, each thread program can be assigned to prove one proof choice. Therefore, the proof generation time could be greatly reduced.

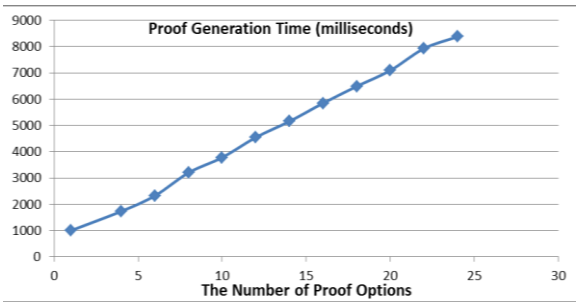

Fig. 6. Proof generation times.

VII. Conclusion

In this paper, we study proof approaches and techniques in a proof-carrying survivability scenario – the system provider constructs a proof and the user verifies that the proof is valid. If so, the system can be considered to satisfy the user's survivability requirements. We show how a system provider chooses a proof choice, collects and compiles survivability property certificates from trusted evaluators, and construct a compliance proof. Our framework can be used to facilitate users to acquire a software system or link a software component to their existing systems real time while ensuring that the external systems will not compromise the survivability of user's existing systems.

References

[1] B. Ellison, D. Fisher, R. Linger, H. Lipson, T. Longstaff, and N. Mead, "Survivable network systems: an emerging discipline," *Technical Report*, Carnegie, Mellon University, November 1997.

[2] V. Westmark, "A definition for information system survivability," *Proc. of the 37th Hawaii International Conference on System Sciences*, Hawaii, USA, 2004.

[3] G. Necula, "Proof-carrying code," *Proc. of 24th ACM SiGPLAN-SIGACT Symposium of Principles of Programming Languages*, 1997, pp. 106-119.

[4] G. Morrisett, D. Walker, K. Crary, and N. Glew, "From system F to typed assembly language," *ACM Transactions on Programming Languages and Systems*, vol. 21, no. 3, pp. 527-568, 1999.

[5] G. Necula and P. Lee, "Safe kernel extensions without run-time checking," *Proc. of Second Symposium on Operating Systems Design and Implementations, Usenix*, 1996, pp. 229-243.

[6] E. Lee and A. Appel, "Policy-enforced linking of entrusted components," *Proc. of ESEC/FSE'03*, Helsinki, Finland, 2003.

[7] L. Bauer, M. Schneider, and E. Felten, "A general and flexible access-control system for the web," *Proceedings of the 11th USENIX Security Symposium*, 2002, pp. 93-108.

[8] D. Garg and F. Pfenning, "A proof-carrying file system," *Proceedings of IEEE Symposium on Security and Privacy*, 2010, pp. 349-364.

[9] N. Li, B. Grosof, and J. Feigenbaum, "Delegation logic: a logic-based approach to distributed authorization," *ACM Transactions on Information and System Security*, vol. 6, no. 1, pp. 128-171, 2003.

[10] Y. Zuo, "A framework of survivability requirement specification for critical information systems," *Proceedings of the 43rd Hawaii International Conference on System Sciences*, Hawaii, USA, 2010.

Yanjun Zuo received a Master's degree and a PhD in Computer Science from the University of Arkansas in Fayetteville, USA. Currently, he is an associate professor of Computer Information Systems at the University of North Dakota, Grand Forks, USA. His research interests include survivable and trustworthy systems, pervasive computing, and information privacy protection. He has published numerous articles in refereed journals, including IEEE Transactions on Secure and Dependable Computing, IEEE Transactions on Systems, Man and Cybernetics, International Journal of Computer and Information Security, Decision Support Systems, and Information System Frontiers.