HAL
archives-ouvertes.fr

A Theory of Integrating Tamper Evidence with Stabilization

Reza Hajisheykhi, Ali Ebnenasir, Sandeep Kulkarni

To cite this version:

Reza Hajisheykhi, Ali Ebnenasir, Sandeep Kulkarni. A Theory of Integrating Tamper Evidence with Stabilization. Mehdi Dastani; Marjan Sirjani. 6th Fundamentals of Software Engineering (FSEN), Apr 2015, Tehran, Iran. Springer, Lecture Notes in Computer Science, LNCS-9392, pp.84-99, 2015, Fundamentals of Software Engineering. <10.1007/978-3-319-24644-4_6>. <hal-01446612>

HAL Id: hal-01446612
https://hal.inria.fr/hal-01446612

Submitted on 26 Jan 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution| 4.0 International License

A Theory of Integrating Tamper Evidence with Stabilization * **

Reza Hajisheykhi[1], Ali Ebnenasir[2], and Sandeep S. Kulkarni[1]

[1] Computer Science and Engineering Department
Michigan State University
East Lansing, Michigan 48824, USA
{hajishey, sandeep}@cse.msu.edu
[2] Department of Computer Science
Michigan Technological University
Houghton, Michigan 49931, USA
aebnenas@mtu.edu

Abstract. We propose the notion of tamper-evident stabilization –that combines stabilization with the concept of tamper evidence– for computing systems. On the first glance, these notions are contradictory; stabilization requires that eventually the system functionality is fully restored whereas tamper evidence requires that the system functionality is permanently degraded in the event of tampering. Tamper-evident stabilization captures the intuition that the system will tolerate perturbation upto a limit. In the event that it is perturbed beyond that limit, it will exhibit permanent evidence of tampering, where it may provide reduced (possibly none) functionality. We compare tamper-evident stabilization with (conventional) stabilization and with active stabilization and propose an approach to verify tamper-evident stabilizing programs in polynomial time. We demonstrate tamper-evident stabilization with two examples and argue how approaches for designing stabilization can be used to design tamper-evident stabilization. We also study issues of composition in tamper-evident stabilization. Finally, we point out how tamper-evident stabilization can effectively be used to provide tradeoff between fault-prevention and fault tolerance.

Keywords: Self-stabilization, reactive systems, adversary, formal methods

1 Introduction

In this paper, we introduce the notion of tamper-evident stabilizing systems, and study these systems in the context of composition, verification, and synthesis. The notion of tamper-evident stabilizing systems is motivated by the need for

* A brief announcement of this paper appears in SSS 2014.
** This work is supported by NSF CCF-1116546, NSF CNS 1329807, and NSF CNS 1318678.

tamper-resistant systems that also stabilize. A tamper-resistant system ensures that an effort to tamper with the system makes the system less useful/inoperable (e.g., by zeroing out sensitive data in a chip or voiding the warranty). The notion of tamper resistance is contradictory to the notion of stabilization in that the notion of stabilization requires that in spite of any possible tampering the system inherently acquires its usefulness eventually.

Intuitively, the notion of tamper-evident stabilization is based on the observation that all tamper-resistant systems tolerate some level of tampering without making the system less useful/inoperable. For example, a tamper-resistant chip may have a circuitry that does some rudimentary checks on the input and discards the input if the check fails. A communication protocol may use CRC to ensure that most random bit-flips in the message are tolerated without affecting the system. However, if the tampering is beyond acceptable level then they become less useful/inoperable. Based on this intuition, we observe that a tamper-evident stabilizing system will recover to its legitimate state if its perturbation is within an acceptable limit. However, if it is perturbed outside this boundary, it will make itself inoperable. Moreover, when the system enters the mode of making itself inoperable, it is necessary that it cannot be prevented.

Thus, if the system is outside its normal legitimate states, it is in one of two modes: *recovery mode*, where it is trying to restore itself to a legitimate state, or *tamper-evident mode*, where it is trying to make itself inoperable. The recovery mode is similar to the typical stabilizing systems in that the recovery should be guaranteed after external perturbations stop. However, in the tamper-evident mode, it is essential that the system makes itself inoperable even if outside perturbations continue.

To realize the last requirement, we need to make certain assumptions about what external perturbations can be performed during tamper-evident mode. For example, if these perturbations could restore the system to a legitimate state then designing tamper-evident stabilizing systems is impossible. Hence, we view the system execution to consist of (1) program executions (in the absence of fault and adversary); (2) program executions in the presence of faults; and (3) program execution in the presence of *adversary.*

Faults are random events that perturb the system randomly and rarely. By contrast, the adversary is *actively* preventing the system from making itself inoperable. However, unlike faults, the adversary may not be able to perturb the system to an arbitrary state. Also, unlike faults, adversary may continue to execute forever. Even if the adversary executes forever, it is necessary that system actions have some fairness during execution. Hence, we assume that the system can make some number (in our formal definitions, we have this as strictly greater than 1) of steps between two steps of the adversary.

The contributions of the paper are as follows. We

- formally define the notion of tamper-evident stabilization;
- compare the notion of tamper-evident stabilization with (conventional) stabilization and active stabilization, where a system stabilizes in spite of the interference of an adversary [7];

– explain the cost of automated verification of tamper-evident stabilization;
– present some theorems about composing tamper-evident stabilizing systems;
– identify how methods for designing stabilizing programs can be used in designing tamper-evident stabilizing systems. We also identify potential obstacles in using those methods, and
– identify potential applications of tamper-evident stabilization and illustrate it with two examples.

Organization. The rest of the paper is organized as follows: In Section 2, we present the preliminary concepts on stabilization. We introduce the notion of tamper-evident stabilization, illustrate it with two examples, and compare it with (conventional) stabilization and active stabilization in Section 3. Section 4 represents an algorithm for automatic verification of tamper-evident stabilizing programs. We evaluate the composition of tamper-evident stabilizing systems in Section 5 and discuss a design methodology for tamper-evident stabilizing programs in Section 6. The relationship between tamper-evident stabilization and other stabilizing techniques is discussed in Section 7, and finally, Section 8 concludes our paper.

2 Preliminaries

Our program modeling utilizes standard approach for defining interleaving programs, stabilization [3, 11, 12], and active stabilization [7]. A program includes a finite set of variables with finite (or any finite abstraction of an infinite state system) domain. It also includes *guarded commands* (a.k.a. *actions*) [11] that update those program variables atomically. Since these internal variables are not needed in the definitions involved in this section, we describe a program in terms of its state space S_p, and its transitions $\delta_p \subseteq S_p \times S_p$, where S_p is obtained by assigning each variable in p a value from its domain.

Definition 1 (Program). *A* program *p is of the form $\langle S_p, \delta_p \rangle$ where S_p is the state space of program p and $\delta_p \subseteq S_p \times S_p$.*

Definition 2 (State Predicate). *A* state predicate *of p is any subset of S_p.*

Definition 3 (Computation). *Let p be a program with state space S_p and transitions δ_p. We say that a sequence $\langle s_0, s_1, s_2, ... \rangle$ is a* computation *iff*

– $\forall j \geq 0 :: (s_j, s_{j+1}) \in \delta_p$

Definition 4 (Closure). *A state predicate S of $p = \langle S_p, \delta_p \rangle$ is* closed *in p iff $\forall s_0, s_1 \in S_p :: (s_0 \in S \wedge (s_0, s_1) \in \delta_p) \Rightarrow (s_1 \in S)$.*

Definition 5 (Invariant). *A state predicate S is an* invariant *of p iff S is closed in p.*

Remark 1. Normally, the definition of invariant (legitimate states) also includes a requirement that computations of p that start from an invariant state are correct with respect to its specification. The theory of tamper-evident stabilization is independent of the behaviors of the program inside legitimate states. Instead, it only focuses on the behavior of p outside its legitimate states. We have defined the invariant in terms of the closure property alone since it is the only relevant property in the definitions/theorems/examples in this paper.

Definition 6 (Convergence). *Let p be a program with state space S_p and transitions δ_p. Let S and T be state predicates of p. We say that T* converges *to S in p iff*

- *$S \subseteq T$,*
- *S is closed in p,*
- *T is closed in p, and*
- *For any computation $\sigma = \langle s_0, s_1, s_2, ...\rangle$ of p if $s_0 \in T$ then there exists l such that $s_l \in S$.*

Definition 7 (Stabilization). *Let p be a program with state space S_p and transitions δ_p. We say that program p is* stabilizing *for invariant S iff S_p converges to S in p.*

Using the approach in [7, 15], we define the adversary as follows and define the notion of tamper-evident stabilization with respect to the capabilities of the given adversary in Section 3.

Definition 8 (Adversary). *We define an* adversary *for program $p = \langle S_p, \delta_p\rangle$ to be a subset of $S_p \times S_p$.*

Next, we define a computation of the program, say p, in the presence of the adversary, say adv.

Definition 9 ($\langle p, adv, k\rangle$-computation). *Let p be a program with state space S_p and transitions δ_p. Let adv be an adversary for program p and k be an integer greater than 1. We say that a sequence $\langle s_0, s_1, s_2, ...\rangle$ is a $\langle p, adv, k\rangle$-*computation *iff*

- $\forall j \geq 0 :: s_j \in S_p,$ *and*
- $\forall j \geq 0 :: (s_j, s_{j+1}) \in \delta_p \cup adv,$ *and*
- $\forall j \geq 0 :: ((s_j, s_{j+1}) \notin \delta_p) \quad \Rightarrow \quad (\forall l \mid j < l < j + k :: (s_l, s_{l+1}) \in \delta_p)$

Observe that a $\langle p, adv, k\rangle$-computation guarantees that there are at least $k - 1$ program transitions/actions between any two adversary actions for $k > 1$. Moreover, the adversary is not required to execute in a $\langle p, adv, k\rangle$-computation.

Remark 2 (Fairness among program transitions). The above definition and definition 3 only consider fairness between program actions and adversary actions. If a program requires fairness among its actions to ensure stabilization, they can be strengthened accordingly. For reasons of space, this issue is outside the scope of this paper.

Definition 10 (Convergence in the presence of adversary). *Let p be a program with state space S_p and transitions δ_p. Let S and T be state predicates of p. Let adv be an adversary for p and let k be an integer greater than 1. We say that T* $\langle adv, k\rangle$-converges *to S in p in the presence of adversary adv iff*

- *$S \subseteq T$,*
- *S is closed in $p \cup adv$,*
- *T is closed in $p \cup adv$, and*
- *For any $\langle p, adv, k\rangle$-computation $\sigma = \langle s_0, s_1, s_2, ...\rangle$ if $s_0 \in T$ then there exists l such that $s_l \in S$.*

Definition 11 (Active stabilization). *Let p be a program with state space S_p and transitions δ_p. Let adv be an adversary for program p and k be an integer greater than 1. We say that program p is* k-active stabilizing *with adversary adv for* invariant *S iff S_p $\langle p, adv, k\rangle$-converges to S in p.*

3 Tamper-Evident Stabilization

This section defines the notion of tamper-evident stabilization, illustrates it in the context of two examples, and compares it with the notion of (conventional) stabilization and active stabilization.

3.1 The Definition of Tamper-Evident Stabilization

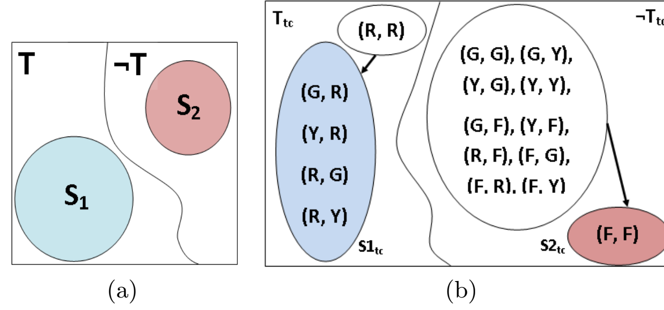
In this section, we define the notion of tamper-evident stabilization.

Definition 12 (Tamper-evident stabilization). *Let p be a program with state space S_p and transitions δ_p. Let adv be an adversary for program p. And, let k be an integer greater than 1. We say that program p is* k-tamper-evident stabilizing *with adversary adv for* invariants $\langle S1, S2\rangle$ *iff there exists a state predicate T of p such that*

- *T converges to $S1$ in p*
- *$\neg T$ $\langle adv, k\rangle$-converges to $S2$ in p.*

From the above definition (especially closure of T and $\neg T$), it follows that $S1$ and $S2$ must be disjoint (See Figure 1(a)). In addition, tamper-evident stabilization provides no guarantees about program behaviors if the adversary executes in T.

Remark 3. Observe that in the above definition k must be greater than 1, as k=1 allows the adversary to prevent the program from executing entirely. In terms of permitted values of k, $k = 2$ provides the maximum power to the adversary. Hence, in most cases, in this paper we will consider k=2. In this case, we will omit the value of k. In other words *tamper-evident stabilizing* is the same as *2-tamper-evident stabilizing.*



(a) (b)

Fig. 1. (a) Structure of a tamper-evident stabilizing system, (b) Tamper-evident stabilizing traffic controller program

Remark 4. Based on the definition of convergence, in the above definition, $S1$ should be a subset of T. Given this constraint, if $S1 = T$ then it corresponds to a *pure tamper evident system*. If such a system is perturbed to a non-legitimate state then it is guaranteed to recover to $S2$ even in the presence of an adversary. And, if $T = S_p$, then it corresponds to a stabilizing program (cf. Theorem 3). Thus, tamper-evident stabilization captures a range of systems from the ones that are *pure tamper-evident* and that are *pure stabilizing*.

The notion of tamper-evident stabilization prescribes the behavior of the program from all possible states. In this respect, it is similar to the notion of stabilizing fault tolerance. In [3], authors introduce the notion of nonmasking fault tolerance; it only prescribes behaviors in a subset of states. We can extend the notion of tamper-evident stabilization in a similar manner. We do so by simply overloading the definition of tamper-evident stabilization.

Definition 13 (Tamper-evident stabilization in environment U). *Let p be a program with state space S_p and transitions δ_p. Let adv be an adversary for program p, and U be a state predicate. Moreover, let k be an integer greater than 1. We say that program p is* k-tamper-evident stabilizing *with adversary adv for* invariants $\langle S1, S2 \rangle$ *in* environment U *iff there exists a state predicate T such that*

- *$S1, S2$, and T are subsets of U,*
- *U is closed in $p \cup adv$,*
- *$U \Rightarrow$ (T converges to $S1$ in p),*
- *$U \Rightarrow$ $\neg T$ $\langle adv, k \rangle$-converges to $S2$ in p.*

Observe that if U equals *true* then the above definition is identical to that of Definition 12.

3.2 The Token Ring Program

This section describes the well-known token ring program [10] and then represent that this program is tamper-evident stabilizing. The program consists of N

processes arranged in a ring. Each process j, $0 \leq j \leq N-1$, has a variable $x.j$ with the domain $\{0, 1, \cdots, N-1\}$. To model the impact of adversary actions on a process j, we add an auxiliary variable $up.j$, where process j has failed iff $up.j$ is false. We say, a process j, $1 \leq j \leq N-1$, has the token iff processes j and $j-1$ have not failed and $x.j \neq x.(j-1)$. If process j, $1 \leq j \leq N-1$, has a token then it copies the value of $x.(j-1)$ to $x.j$. The process 0 has the token iff processes 0 and $N-1$ have not failed and $x.(N-1) = x.0$. If process 0 has the token then it increments its value in modulo N arithmetic (we show modulo N arithmetic by notation $+_N$). Thus, the actions of the program are as follows:

$$\begin{array}{lll} TR_0 :: & up.0 \wedge up.(N-1) \wedge x.0 = x.(N-1) & \longrightarrow \quad x.0 := (x.(N-1) +_N 1) \\ TR_j :: & up.j \wedge up.(j-1) \wedge x.j \neq x.(j-1) & \longrightarrow \quad x.j := x.(j-1); \end{array}$$

Adversary action. The adversary can cause any process to fail. Hence, the adversary action can be represented as

$$TR_{adv} :: up.j \quad \longrightarrow up.j := false$$

Tamper-evident stabilization of the program. To show that the token ring program TR is *tamper-evident stabilizing* in the presence of the adversary TR_{adv}, we define the predicate T_{tr} and invariants $S1_{tr}$ and $S2_{tr}$ as follows:

$$\begin{array}{lll} T_{tr} & = & \forall j :: up.j \\ S1_{tr} & = & T_{tr} \ \wedge (\forall j : 1 \leq j \leq N-1 : (x.j = x.(j-1)) \vee (x.(j-1) = x.j +_N 1)) \\ & & \quad \wedge ((x.0 = x.(N-1)) \vee (x.0 = x.(N-1) +_N 1)) \\ S2_{tr} & = & \neg T_{tr} \wedge (\forall j : 1 \leq j \leq N-1 : (up.j \wedge up.(j-1)) \Rightarrow x.j = x.(j-1)) \\ & & \quad \wedge ((up.0 \wedge up.(N-1)) \Rightarrow (x.0 \neq x.(N-1))) \end{array}$$

Theorem 1. *The token ring program TR is* tamper-evident stabilizing *with adversary TR_{adv} for* invariants $\langle S1_{tr}, S2_{tr} \rangle$.

Proof. If T_{tr} is true then the program is essentially the same as the token ring program from [11] and, hence, it stabilizes to $S1_{tr}$. If T_{tr} is violated then the token cannot go past failed process(es). Hence, $S2_{tr}$ would eventually be satisfied. Note that for the second constraint, adversary action (that may fail a process) cannot prevent the program from reaching $S2_{tr}$. □

3.3 Tamper-Evident Stabilizing Traffic Controller Program

This section describes another tamper-evident stabilizing program that illustrates a traffic light program that (1) recovers to normal operation from perturbations that do not cause the system to reach an unsafe state, and (2) permanently preserves the evidence of tampering if perturbations cause the system to reach an unsafe state. This example also illustrates why tamper-evident stabilization is desirable over (conventional) stabilization in some circumstances. Moreover, it can be used as a part of multiphase recovery [6] where a quick recovery is provided to safe states and complete recovery to legitimate states can be obtained later (or with human intervention).

Description of the program. In this program, we have an intersection with two one-way roads [5]. Each road is associated with a signal that can be either green (G), yellow (Y), red (R), or flashing (F). As expected, in any normal state, at least one of the signals should be red to ensure that traffic accidents do not occur.

If such a system is perturbed by an adversary where an adversary can somehow affect the signal operation causing safety violations then it is crucial that such an occurrence is noted for potential investigation. (These adversary actions can be triggered with simple transient faults that reset clock variables. For simplicity, we omit the cause of such adversary actions and only consider their effects.) In this example, we consider the requirement that if both signals are simultaneously yellow or green then the system must reach a state where both signals are flashing to indicate a signal malfunction due to adversary.

Thus, this program consists of two variables sig_0 and sig_1. The program consists of five actions: The first two actions are responsible for normal operation where a signal changes from G to Y to R and back to G. The third action considers the case where the system is perturbed outside legitimate states (e.g., by transient faults) and it is desirable that the system recovers from that state. The fourth action considers the case where the adversary actions perturb the system beyond an acceptable level and, hence, it is necessary that the system enters the tamper-evident state. Thus, the program actions are as follows: (In this program, j is instantiated to be either 0 or 1, and k is instantiated to be $1-j$.)

$TC1_j :: (sig_j = G) \wedge (sig_k = R) \longrightarrow sig_j = Y$
$TC2_j :: (sig_j = Y) \wedge (sig_k = R) \longrightarrow (sig_j = R) \wedge (sig_k = G)$
$TC3_j :: (sig_j = R) \wedge (sig_k = R) \longrightarrow (sig_j = G)$
$TC4_j :: ((sig_j \neq R) \wedge (sig_k \neq R)) \vee (sig_k = F) \longrightarrow (sig_j = F)$
$TC5_j :: (sig_j = F) \wedge (sig_k = F) \longrightarrow \{$*notify the user that the system is in S2*$\}$

Adversary actions. The adversary TC_{adv} can cause a red signal to become either yellow or green. Hence, the adversary actions can be represented as ($j = 0, 1$):

$TC_{adv_1} :: sig_j = R \longrightarrow sig_j = Y$
$TC_{adv_2} :: sig_j = R \longrightarrow sig_j = G$

Tamper-evident stabilization of the program. To show that the program TC is tamper-evident stabilizing in the presence of adversary TC_{adv}, we define the predicate T_{tc} and invariants $S1_{tc}$ and $S2_{tc}$ as follows:

$T_{tc} = \langle((G,R),(Y,R),(R,G),(R,Y)),(R,R)\rangle$
$S1_{tc} = \langle(G,R),(Y,R),(R,G),(R,Y)\rangle$
$S2_{tc} = \langle(F,F)\rangle$

Theorem 2. *The traffic controller program TC is* tamper-evident stabilizing *with adversary TC_{adv} for* invariants $\langle S1_{tc}, S2_{tc}\rangle$.

Proof. If T_{tc} is true then the program is essentially the same as the traffic control program from [5] and, hence, it stabilizes to $S1_{tc}$. If the adversary TC_{adv} violates T_{tc}, the action $TC4$ can execute and one of the signals will be flashing. As a result, the other signal would eventually become flashing and $S2_{tr}$ would be satisfied (See Figure 1(b)). □

3.4 Stabilization, Tamper-evident Stabilization, and Active Stabilization

In this section, we compare the notion of (conventional) stabilization, active stabilization and tamper-evident stabilization. Specifically, Theorem 3 considers the case where p is stabilizing and evaluates whether it is tamper-evident stabilizing, and Theorem 4 considers the reverse direction. Relation with active stabilization follows trivially from these theorems.

Theorem 3. *If a program p is* stabilizing *for* invariant S*, then p is* k-tamper-evident stabilizing *with adversary adv for* invariants $\langle S, \emptyset \rangle$*, for any adversary adv and $k \geq 2$.*

Proof. To prove tamper-evident stabilization, we need to identify a value of T. We set $T = true$, representing the state space of p. Now, we need to show that S_p converges to S in p and $\neg true$ $\langle adv, k \rangle$-converges to ϕ in p. Of these, the former is satisfied since p is stabilizing for invariant S, and the latter is trivially satisfied since $\neg true$ corresponds to the empty set. □

Corollary 1. *If program p is* k-active stabilizing *with adversary adv and $k \geq 2$ for* invariant S*, then p is* k-tamper-evident stabilizing *with adversary adv for* invariants $\langle S, \emptyset \rangle$.

Note that, if there exists k and adv such that program p is k-active stabilizing with adversary adv for invariant S, then p is stabilizing for invariant S.

Theorem 4. *If program $p = \langle S_p, \delta_p \rangle$ is* k-tamper-evident stabilizing *with adversary adv for* invariants $\langle S1, S2 \rangle$*, then p is* stabilizing *for* invariant $(S1 \vee S2)$.

Proof. Since program p is tamper-evident stabilizing, the two constraints in the definition of tamper-evident stabilizing are true. If the program p starts from T, it converges to $S1$. If p starts from $\neg T$, in the presence or absence of adversary adv, it converges to $S2$. This completes the proof. □

However, a similar result relating tamper-evident stabilization and active stabilization is not valid. In other words, it is possible to have a program p that is *k-tamper-evident stabilizing* with adversary adv for *invariants* $\langle S1, S2 \rangle$ but it is not *k-active stabilizing* with adversary adv for *invariant* $(S1 \vee S2)$. This is due to the fact that if the program begins in T then in the presence of the adversary, there is no guarantee that it would recover to $S1$.

4 Verification of Tamper-evident Stabilization

To prove tamper-evident stabilization of a given program, we need to determine the predicate T (from Definition 12). Based on Definition 12, from every state in $\neg T$, we must eventually reach a state in $S2$. Hence, from $\neg T$, we cannot reach a state in $S1$. Also, from every state in T, we must reach a state in $S1$. Thus, the only possible choice for T is the states from where the program can reach $S1$. Therefore, Algorithm 1 starts with the construction of T (Lines 1-3) and checking the closure property of predicates T and $\neg T$, and invariants $S1$ and $S2$ (Lines 4-6). Thereafter, we utilize CheckCycle() to detect if program p has cycles in $T - S1$. Notice that if there is a cycle in a state predicate Y, then the following is true for any state s_0 in the cycle: $\exists s_1 \in Y : (s_0, s_1) \in p$. As such, the absence of any cycles in Y would require the negation of the aforementioned expression to hold (see Line 16). This is the basic idea behind the CheckCycle routine (Lines 15-19). If any states in $T - S1$ is not removed, it implies that some of them form a cycle. If such a cycle exists then p is not tamper-evident stabilizing.

Utilizing the ideas in [7], we construct p_1 that considers the effect of adversary adv and checks for cycles of p_1 in $\neg T - S2$ (Line 8-9). In this construction, $reach(s_0, s_1, l)$ denotes that s_1 can be reached from s_0 by execution of exactly l transitions of $\neg T$. If such cycles of p_1 do not exist then p is tamper-evident stabilizing.

Algorithm 1 Verification of tamper-evident stabilization

Input: program $p = \langle S_p, \delta_p \rangle$, invariants $S1$ and $S2$, adversary adv.
Output: *true* or *false*.

1: $T = S1$
2: **repeat** $T1 = T$; $T = T1 \cup \{s_0 \mid (s_0, s_1) \in \delta_p \ \wedge \ s_1 \in T\}$
3: **until** $(T1 == T)$
4: **if** $\neg$(CheckClosure(T, p) $\wedge$ CheckClosure($\neg T$, p) $\wedge$ CheckClosure($S1$, p) $\wedge$ CheckClosure($S2$, p)) **then**
5:     **return** *false*
6: **end if**
7: **if** CheckCycle($T - S1, p$) $\neq \emptyset$ **then** **return** *false* **end if**
8: $p_1 = \{(s_0, s_1) \mid (\exists l : l \geq k-1 : reach(s_0, s_1, l)) \ \vee \ (\exists s_2 : reach(s_0, s_2, l) \wedge (s_2, s_1) \in adv)\}$
9: **if** CheckCycle($\neg T - S2, p_1$) $\neq \emptyset$ **then** **return** *false* **end if**
10: **return** *true*

11: **function** CheckClosure(X, p)
12:     **if** $\forall s_0, s_1 \in S_p : (s_0 \in X \wedge (s_0, s_1) \in \delta_p) \ \Rightarrow \ (s_1 \in X)$ **then** **return** *true*
13:     **else** **return** *false* **end if**
14: **end function**
15: **function** CheckCycle(Y, p)
16:     **repeat** $Y1 = Y$; $Y = Y_1 - \{s_0 \mid \forall s_1 \in Y : (s_0, s_1) \notin \delta_p\}$
17:     **until** $(Y1 == Y)$
18:     **return** Y
19: **end function**

Theorem 5. *The following problem can be solved in polynomial time in* $|S_p|$.[3]

[3] For reason of space, proofs appear in [17].

Given a program p, adversary adv, and state predicates $S1$ and $S2$, is p tamper-evident stabilizing with adversary adv for invariants $\langle S1, S2\rangle$?

5 Composing Tamper-evident Stabilization

In this section, we evaluate the composition of tamper-evident stabilizing systems by investigating different types of compositions considered for stabilizing systems.

Parallel Composition. A parallel composition of two programs considers the case where two independent programs are run in parallel on a weakly fair scheduler so that each program is guaranteed to execute its enabled actions. Weak fairness ensures that any action that is continuously enabled will be executed infinitely often. Thus, during the parallel execution, the behavior of one program does not affect the behavior of the other. Hence, if we have two programs p and q that do not share any variables such that p is stabilizing for S and q is stabilizing for R then parallel composition of p and q is stabilizing for $S \wedge R$.

Now, we consider the case where we have two programs p and q that are tamper-evident stabilizing for $\langle S1, S2\rangle$ and $\langle S1', S2'\rangle$, and p and q do not share any variables. Is the parallel composition of p and q (denoted by $p[]q$) also tamper-evident stabilizing?

Theorem 6 (Parallel Composition). *Given programs p and q that do not share variables.*

p is tamper-evident stabilizing with adversary adv for $\langle S1, S2\rangle$ $\wedge$
q is tamper-evident stabilizing with adversary adv for $\langle S1', S2'\rangle$
$\Rightarrow$
$p[]q$ is tamper-evident stabilizing with adversary adv for $\langle S1 \wedge S1', S2 \vee S2'\rangle$

Note that in parallel composition of two tamper-evident stabilizing programs, the first predicate is combined by conjunction whereas the second one is combined by disjunction. However, we could make $p[]q$ tamper-evident stabilizing for $\langle S1 \wedge S1', S2 \wedge S2'\rangle$ provided we add actions to p (respectively q) so that it checks if q (respectively, p) is in a state in $S2'$ (respectively, $S2$). Accordingly, p can change its own state to be in $S2$ (respectively, $S2'$).

Superposition. We can also superpose two tamper-evident stabilizing systems in a similar manner. For example, consider the case where program p is superposed on program q, i.e., p has read-only access to variables of q and q does not have access to variables of p.

Theorem 7 (Superposition).

p is tamper-evident stabilizing with adversary adv for $\langle S1, S2\rangle$ in $S1'$ $\wedge$
q is active stabilizing with adversary adv for $S1'$ $\wedge$
q is tamper-evident stabilizing with adversary adv for $\langle S1', S2'\rangle$ $\wedge$
q is silent in $S1'$, i.e., q has no transition (except self-loops) in $S1'$ $\wedge$

p is superposed on q
$\Rightarrow$
$p[]q$ *is tamper-evident stabilizing with adversary adv for* $\langle S1, S2 \vee S2' \rangle$.

Transitivity. Tamper-evident stabilization preserves transitivity in a manner similar to stabilizing programs. Specifically,

Theorem 8 (Transitivity 1).

p is tamper-evident stabilizing with adversary adv for $\langle S1, S2 \rangle$ *in* $U \wedge$
p is tamper-evident stabilizing with adversary adv for $\langle S1', S2' \rangle$ *in* $S1$
$\Rightarrow$
p is tamper-evident stabilizing with adversary adv for $\langle S1', S2 \rangle$ *in* U*, and*
p is tamper-evident stabilizing with adversary adv for $\langle S1', S2 \vee S2' \rangle$ *in* U.

We can also infer transitivity property by the following theorem.

Theorem 9 (Transitivity 2).

p is tamper-evident stabilizing with adversary adv for $\langle S1, S2 \rangle \wedge$
$S1$ *converges to* $S1'$ *in* $p \wedge$
$S2$ $\langle adv, k \rangle$*-converges to* $S2'$ *in* p
$\Rightarrow$
p is tamper-evident stabilizing with adversary adv for $\langle S1', S2' \rangle$.

6 Designing Tamper-evident Stabilization by Local Detection and Global/Local Correction

In this section, we identify some possible approaches for designing tamper-evident stabilization. Specifically, we evaluate the use of some of the existing approaches for designing stabilization in designing tamper-evident stabilization.

Local Detection and Global Correction One approach for designing stabilization is via local detection and global correction. In such a system, the invariant S of the system is of the form $\forall j : S.j$, where $S.j$ is a local predicate that can be checked by process j. Each process j is responsible for checking its own predicate. If the system is outside the legitimate state then the local predicate of at least one process is violated. Hence, this process is responsible for initiating a global correction (such as distributed reset [19]) to restore the system to a legitimate state.

A similar approach is also applicable for tamper-evident stabilization. For example, consider the case where the predicates involved in defining tamper-evident stabilization are $S1 = \forall j :: S1.j$, $S2 = \forall j :: S2.j$, and $T = \forall j :: T.j$. Based on the problem of tamper-evident stabilization, we have $\forall j :: (S1.j \Rightarrow T.j) \wedge (S2.j \Rightarrow \neg T.j) \wedge \neg(S1.j \wedge S2.j)$.

In this case, the actions of process j to obtain tamper-evident stabilization is as follows:

$\neg T.j \;\wedge\; \neg S2.j \longrightarrow$ *Satisfy* $S2.j$

$T.j \ \wedge \ \neg S1.j \longrightarrow$ *Initiate global correction to restore* $S1$

To utilize such an approach to design tamper-evident stabilization, we need to make some changes to global correction and put some reasonable constraints on what an adversary can do. In particular, the global correction to restore $S1$ involves changes to all processes. For tamper-evident stabilization, however, process j will execute its part in global correction only if $T.j$ is true. Also, if process j observes that $T.k$ is false for some neighbor k then j will satisfy $S2.j$. This will guarantee that if $T.j$ is false for some process then the program will eventually reach a state in $S2$. The definition of tamper-evident stabilization requires that $\neg T$ is closed in the adversary actions. This assumption is essential since if the adversary could move the system from a state in $\neg T$ to T then the system would have forgotten that it was tampered beyond acceptable levels. In the context of this example, it would be necessary that the adversary cannot cause the program to start in a state where $T.j$ is false for some process j and the adversary causes j to move to a state where $T.j$ is true.

Local Detection and Local Correction We can also utilize the above approach in the context of local detection and local correction [3] to add tamper-evident-stabilization if invariant $S1$ is of the form $\forall j :: S1.j$, predicates of different processes are arranged in a partial order, and actions that correct $S1.j$ preserve all predicates that come earlier in the order. In such a system when process j finds that $T.j \wedge \neg S1.j$ is true it only locally satisfies $S1.j$. Given that we have a partial order, eventually we reach a state where $S.j$ is true in all states.

Effect of the structure of the predicate T. Intuitively, in tamper-evident stabilization, we have two convergence requirements. T converges to $S1$ and $\neg T$ converges to $S2$ in the presence of an adversary. If T is a conjunctive predicate then $\neg T$ is a disjunctive predicate. Hence, a reader may wonder what would happen if T were a disjunctive predicate instead of a conjunctive predicate. We argue that this is likely to be a harder problem than the case where T is a conjunctive predicate.

7 The Relationship between Tamper-evident Stabilization and other Stabilization Techniques

Starting with Dijkstra's seminal work [10] on stabilizing algorithms for token circulation, several variations of stabilizing algorithms have been proposed during the past decades. These algorithms can be classified into two categories: *stronger* stabilizing and *weaker* stabilizing algorithms.

The algorithms in the first category not only guarantee stabilization but also satisfy some additional properties. Examples of this category include fault-containment stabilization, byzantine stabilization, Fault-Tolerant Self Stabilization (FTSS), multitolerance, and active stabilization. Fault-containment stabilization (e.g., [14, 25]) refers to stabilizing programs that ensure that if one (respectively small number of) fault occurs then quick recovery is provided to the invariant. Byzantine stabilizing (e.g., [21, 22]) programs tolerate the sce-

narios where a subset of processes is byzantine. FTSS (e.g., [4]) covers stabilizing programs that tolerate permanent crash faults. Multitolerant stabilizing (e.g., [13, 19]) systems ensure that, in addition to stabilization, the program masks a certain class of faults. Finally, active stabilization [7] requires that the program should recover to the invariant even if it is constantly perturbed by an adversary.

By contrast, a stabilizing program satisfies the constraints of weaker versions of stabilization. However, a program that provides a weaker version of stabilization may not be stabilizing. Examples of this include weak stabilization, probabilistic stabilization, and pseudo stabilization. Weak stabilization (e.g., [9, 16]) requires that starting from any initial configuration, there exists an execution that eventually reaches a point from which its behavior is correct. However, the program may execute on a path where such a legitimate state is never reached. Probabilistic stabilization [18] refers to problems that ensure that starting from any initial configuration, the program converges to its legitimate states with probability 1. Nonmasking fault tolerance (e.g., [1,2]) targets the programs where the program recovers from states reached in the presence of a limited class of faults. However, this limited set of states may not cover the set of all states. Pseudo stabilization [8] relaxes the notion of points in the execution from which the behavior is correct. In other words, every execution has a suffix that exhibits correct behavior, yet time before reaching this suffix is unbounded.

The aforementioned stabilizing algorithms consider several problems including mutual exclusion, leader election, consensus, graph coloring, clustering, routing, and overlay construction. However, none of them considers problem of tampering (e.g., [20, 23, 24]). In part, this is due to the fact that stabilization and tamper evidence are potentially conflicting requirements.

Tamper-evident stabilization is in some sense a weaker version of stabilization in that from Theorem 3 every stabilizing program is also tamper-evident stabilizing. In particular, a stabilizing program guarantees that from all states program would eventually recover to legitimate states. By contrast, tamper-evident stabilizing program gives the option of recovering to *tamper-evident* states. (Although Theorem 4 suggests that every tamper-evident stabilizing program can be thought of as a stabilizing program, the invariant of such a stabilizing program is of the form $\langle S1, S2 \rangle$, where $S2$ includes states that the system has no/reduced functionality.)

Tamper-evident stabilization is stronger than the notion of nonmasking fault tolerance. In particular, nonmasking fault-tolerance also has the notion of fault-span (similar to T in Definition 12) from where recovery to the invariant is provided. In tamper-evident stabilization, if the program reaches a state in $\neg T$, it is required that it stays in $\neg T$. By contrast, in nonmasking fault-tolerance, the program *may* recover from $\neg T$ to T.

Tamper-evident stabilization can be considered as a special case of nonmasking-failsafe multitolerance, where a program that is subject to two types of faults F_f and F_n provides (i) failsafe fault tolerance when F_f occurs, (ii) nonmasking tolerance in the presence of F_n, and (iii) no guarantees if both F_f and F_n occur

in the same computation. We have previously identified [13] sufficient conditions for efficient stepwise design of failsafe-nonmasking multitolerant systems, where F_f and F_n do not occur simultaneously and their scopes of perturbation outside the invariant are disjoint. Based on the role of T in Definition 12, we can ensure that these conditions are satisfied (Due to reasons of space, this proof is beyond the scope of the paper) for tamper-evident stabilization. This suggests that efficient algorithms can be designed for tamper-evident stabilization based on the approach in [13].

8 Conclusion and Future Work

This paper introduces the notion of *tamper-evident stabilization* that captures the requirement that if a system is perturbed within an acceptable limit then it restores itself to legitimate states. However, if it is perturbed beyond this boundary then it permanently preserves evidence of tampering. Moreover, the latter operation is unaffected even if the adversary attempts to stop it. We formally defined tamper-evident stabilization and investigated how it relates to stabilization and active stabilization. We argued that tamper-evident stabilization is weaker than stabilization in that every stabilizing system is indeed tamper-evident stabilizing. Also, tamper-evident stabilization captures a spectrum of systems from *pure tamper-evident systems* to *pure stabilizing systems*. We also demonstrated two examples where we design tamper-evident stabilizing token passing and traffic control protocols. We identified how methods for designing stabilizing programs can be leveraged to design tamper-evident stabilizing programs. We showed that the problem of verifying whether a given program is tamper-evident stabilizing is polynomial in the state space of the given program. We note that the problem of adding tamper-evident stabilization to a given high atomicity program can be solved in polynomial time. However, the problem is NP-hard for distributed programs. Moreover, we find that parallel composition of tamper-evident stabilizing systems works in a manner similar to that of stabilizing systems. Nevertheless, superposition or transitivity requirements of tamper-evident stabilization are somewhat different than that for stabilizing systems.

We are currently investigating the design and analysis of tamper-evident stabilizing System-on-Chip (SoC) systems in the context of the IEEE SystemC language. Our objective here is to design systems that facilitate reasoning about what they do and what they do not do in the event of tampering. Second, we will leverage our existing work on model repair and synthesis of stabilization in automated design of tamper-evident stabilization. Third, we plan to study the application of tamper-evident stabilization in game theory (and vice versa).

References

1. A. Arora. Efficient reconfiguration of trees: A case study in methodical design of nonmasking fault-tolerant programs. In *FTRTFT*, pages 110–127, 1994.

2. A. Arora, M. Gouda, and G. Varghese. Constraint satisfaction as a basis for designing nonmasking fault-tolerant systems. *Journal of High Speed Networks*, 5(3):293–306, 1996.
3. A. Arora and M. G. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19(11):1015–1027, 1993.
4. J. Beauquier and S. Kekkonen-Moneta. On ftss-solvable distributed problems. In *WSS*, pages 64–79, 1997.
5. B. Bonakdarpour and S. S. Kulkarni. Compositional verification of fault-tolerant real-time programs. In *EMSOFT*, pages 29–38, 2009.
6. B. Bonakdarpour and S. S. Kulkarni. On the complexity of synthesizing relaxed and graceful bounded-time 2-phase recovery. In *FM*, pages 660–675, 2009.
7. B. Bonakdarpour and S. S. Kulkarni. Active stabilization. In *SSS*, pages 77–91, 2011.
8. J.E. Burns, M. Gouda, and R.E. Miller. Stabilization and pseudo-stabilization. *Distributed Computing*, 7(1):35–42, 1993.
9. S. Devismes, S. Tixeuil, and M. Yamashita. Weak vs. self vs. probabilistic stabilization. In *ICDCS*, pages 681–688, 2008.
10. E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.
11. E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1990.
12. S. Dolev. *Self-Stabilization.* MIT Press, 2000.
13. A. Ebnenasir and S. S. Kulkarni. Feasibility of stepwise design of multitolerant programs. *TOSEM*, 21(1):1–49, December 2011.
14. S. Ghosh and A. Gupta. An exercise in fault-containment: Self-stabilizing leader election. *Information Processing Letters*, 59(5):281–288, 1996.
15. M. Gouda. Elements of security: Closure, convergence, and protection. *Information Processing Letters*, 77(24):109 – 114, 2001. In honor of Edsger W. Dijkstra.
16. M. Gouda. The theory of weak stabilization. In *International Workshop on Self-Stabilizing Systems*, volume 2194, pages 114–123, 2001.
17. R. Hajisheykhi, A. Ebnenasir, and S. Kulkarni. Tamper-evident stabilization. Technical Report MSU-CSE-14-4, June 2014.
18. A. Israeli and M. Jalfon. Token management schemes and random walks yield self-stabilizing mutual exclusion. In *PODC*, pages 119–131, 1990.
19. S. Kulkarni and A. Arora. Multitolerance in distributed reset. *Chicago Journal of Theoretical Computer Science*, 1998(4), December 1998.
20. D. Lie, C. A. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. C. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *ASPLOS*, pages 168–177, 2000.
21. Mahyar R. Malekpour. A byzantine-fault tolerant self-stabilizing protocol for distributed clock synchronization systems. In *SSS*, pages 411–427, 2006.
22. M. Nesterenko and A. Arora. Tolerance to unbounded byzantine faults. In *SRDS*, pages 22–31, 2002.
23. Sean W. Smith and Steve Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks*, 31(8):831–860, 1999.
24. G. E. Suh, D. E. Clarke, B. Gassend, M. van Dijk, and S. Devadas. Aegis: architecture for tamper-evident and tamper-resistant processing. In *ICS*, pages 160–171, 2003.
25. H. Zhang and A. Arora. Guaranteed fault containment and local stabilization in routing. *Computer Networks*, 50(18):3585–3607, 2006.