Science of Computer Programming 160 (2018) 93–114

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

A theory of integrating tamper evidence with stabilization

Reza Hajisheykhi [a,*], Ali Ebnenasir [b], Sandeep S. Kulkarni [a]

[a] Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824, USA
[b] Department of Computer Science, Michigan Technological University, Houghton, MI 49931, USA

ARTICLE INFO

Article history:
Received 16 February 2016
Received in revised form 6 December 2016
Accepted 6 March 2017
Available online 21 March 2017

Keywords:
Self-stabilization
Reactive systems
Adversary
Formal methods

ABSTRACT

We propose the notions of tamper-evident stabilization and flexible tamper-evident stabilization – that combine 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 and flexible tamper-evident stabilization capture the intuition that the system will tolerate perturbations 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 two approaches to verify tamper-evident and flexible tamper-evident stabilizing programs in polynomial time in the size of state space. We demonstrate tamper-evident stabilization with two examples and point out some of its potential applications. We also demonstrate how approaches for designing stabilization can be used to design tamper-evident and flexible tamper-evident stabilizations. Finally, we study issues of composition in tamper-evident and flexible tamper-evident stabilizations and discuss how tamper-evident stabilization can effectively be used to provide tradeoff between fault-prevention and fault tolerance.

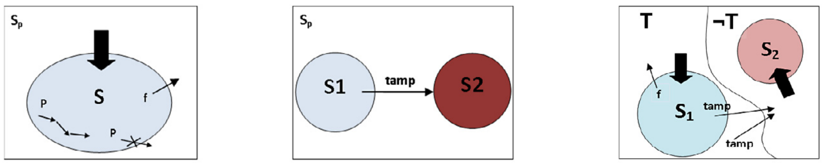
1. Introduction

A *stabilizing* system [15] ensures that it will recover to a set of legitimate states (S in Fig. 1(a), a.k.a invariant) even if it starts executing from an arbitrary state. For this reason, stabilization is often utilized as a type of fault-tolerance that requires recovery from unexpected transient faults. In other words, if the faults perturb the system to an arbitrary state, the goal of stabilizing systems is to ensure that the system will recover to the legitimate states with the assumption that no additional faults will occur. Nevertheless, stabilizing systems may not recover to legitimate states in the presence of tampering. Tamper-resistant systems are mostly utilized for secure chip designs (e.g. [37]). While *tamper-resistant* systems can prevent tampering to some degree, if tampering cannot be prevented, the system reaches a set of states where in the system is less functional/inoperable ($S2$ in Fig. 1(b)).

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. Therefore, in this paper, we combine these two seemingly conflicting concepts to benefit from the advantages of both. Specifically, we introduce the

* Corresponding author.
E-mail addresses: hajishey@cse.msu.edu (R. Hajisheykhi), aebnenas@mtu.edu (A. Ebnenasir), sandeep@cse.msu.edu (S.S. Kulkarni).

http://dx.doi.org/10.1016/j.scico.2017.03.001
0167-6423/© 2017 Elsevier B.V. All rights reserved.



(a) Stabilization (b) Tamper-resistance (c) Tamper-evident stabilization

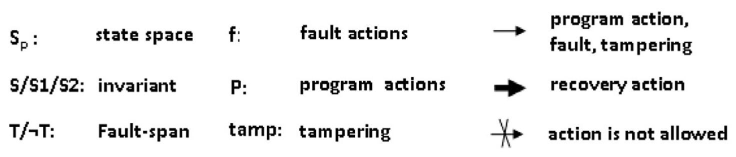


Fig. 1. Stabilization vs. tamper-resistance.

notions of tamper-evident stabilizing and flexible tamper-evident stabilizing systems, and identify their properties in terms of composition, verification, and synthesis. The notions of tamper-evident and flexible tamper-evident stabilizing systems are motivated by the need for 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).

Intuitively, the notions of tamper-evident and flexible tamper-evident stabilizations are 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 some acceptable level then they become less useful/inoperable (see Fig. 1(b)). Based on this intuition, we observe that tamper-evident and flexible tamper-evident stabilizing systems will recover to their legitimate states if their perturbation is within an acceptable limit. However, if they are perturbed outside this boundary, they will make themselves inoperable. Moreover, when the systems enter the mode of making themselves inoperable, it is necessary that it cannot be prevented.

Thus, if a tamper-evident stabilizing 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 (the T area in Fig. 1(c)), or *tamper-evident mode*, where it is trying to make itself inoperable (the $\neg T$ area in Fig. 1(c)). 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 would be 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 of steps between two steps of the adversary (in our formal definitions, we have this number of steps as strictly greater than 1).

Moreover, the notion of flexible tamper-evident stabilization is a more general definition of tamper-evident stabilization. Specifically, if a flexible tamper-evident stabilizing system is outside its legitimate states, it is in one of the recovery, tamper-evident, or *boundary* modes. The first two modes are similar to those in tamper-evident stabilizing systems. In the boundary mode, while the system can recover to either the set of legitimate states or tamper-evident states, an authorized operator decides to which set of states recovery should be achieved (depending on other environmental knowledge unavailable to the system).

The contributions of this paper are as follows. We

- formally define the notions of tamper-evident stabilization and flexible 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 [12];
- present algorithms for the verification of tamper-evident stabilization and flexible tamper-evident stabilization;
- identify potential applications of tamper-evident stabilization and illustrate it with three examples;
- present some theorems about composing tamper-evident stabilizing and flexible tamper-evident stabilizing systems, and

- 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.

Organization. The rest of the paper is organized as follows: Section 2 presents the preliminary concepts on stabilization. Section 3 introduces the notion of tamper-evident stabilization and illustrates it with an example. Section 3.2 provides another tamper-evident stabilizing program that demonstrates why tamper-evident stabilization is desirable over (conventional) stabilization. Section 4 compares tamper-evident stabilization with (conventional) stabilization and active stabilization. Section 5 represents an algorithm for verification of tamper-evident stabilizing programs. Section 6 explains several potential applications for tamper-evident stabilization. Section 7 introduces the notion of flexible tamper-evident stabilization, describes its relation with tamper-evident stabilization, and proposes an approach for verification of flexible tamper-evident stabilizing programs. Section 8 evaluates the composition of tamper-evident and flexible tamper-evident stabilizing systems. Section 9 describes a design methodology for tamper-evident stabilizing programs. The relationship between tamper-evident stabilization and other stabilizing techniques is discussed in Section 10. Finally, Section 11 concludes our paper.

2. Preliminaries

Our program modeling utilizes standard approach for defining interleaving programs, stabilization [16,5,17], and active stabilization [12]. A program includes (1) a finite set of variables with (finite or any finite abstraction of an infinite state system) domain, and (2) a set of *guarded commands* (a.k.a. *actions*) [16] that update those program variables atomically. Since these internal variables are not needed in the definitions involved in this section, we describe a program, say p, 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 program 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 $\delta_p \subseteq S_p \times S_p$.

Assumption 1. Given a program p, if some state, say s, in S_p has no outgoing transition then we add transition (s, s). The rest of the definitions in this paper are with respect to a program that is modified in this fashion. The purpose of this change is only to simplify subsequent definitions by obviating the need to consider terminating behaviors explicitly. It does not restrict programs.

Definition 2 *(State predicate).* A *state predicate* of a program 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 a program $p = \langle S_p, \delta_p \rangle$ 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 a program 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 program behaviors 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.

Definition 8 *(Faults). Faults* for program $p = \langle S_p, \delta_p \rangle$ is a subset of $S_p \times S_p$; i.e., the faults can perturb the program to any arbitrary state.

The *adversary*, say adv, for the program p is defined in terms of its transitions. This allows us to model the limited set of actions the adversary may be allowed to execute. A typical stabilizing program in the literature considers the case where faults perturb the system to an arbitrary state. Subsequently, the goal of the program is to ensure that the system will recover to a legitimate state with the assumption that no additional faults will occur. However, unlike faults, the adversary actions may never stop. In particular, it would be unreasonable to assume that the adversary actions stop for a long enough time for the system to stabilize. Using the approach in [12,26], we define the adversary as follows and define the notion of tamper-evident stabilization with respect to the capabilities of the given adversary.

Definition 9 *(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 10 *($\langle p, adv, k \rangle$-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

- $\forall j \geq 0 :: s_j \in S_p$,
- $\forall j \geq 0 :: (s_j, s_{j+1}) \in \delta_p \cup adv$, 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. This issue is outside the scope of this paper.

Definition 11 *($\langle p, adv, k \rangle$-convergence).* 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 (i.e. $T \langle p, adv, k \rangle$-converges to S) 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 σ (=$\langle s_0, s_1, s_2, ...\rangle$) if $s_0 \in T$ then there exists l such that $s_l \in S$.

Observe that, in Definition 7, the program converges to its legitimate states in the presence of faults. However, if a self-stabilizing program is perturbed by an adversary continuously, it may not converge to its legitimate states. Active stabilizing systems consider the adversary actions too and converge to legitimate states even if they are perturbed by the adversary continuously. Hence, utilizing Definition 11, we define active stabilization as follows [12].

Definition 12 *(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 *strong 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

In this section, we formally define tamper-evident stabilization utilizing the aforementioned definitions in Section 2 and illustrate it in the context of the well-known token ring program [15]. Specifically, we define a program that converges (1) to its legitimate states in the presence of faults, and (2) to a set of states, called *tamper-evident states*, if it is perturbed by an adversary continuously. We call the set of legitimate states $S1$ and the set of tamper-evident states $S2$. Additionally, we define a boundary T upto which the program will recover to its legitimate states $S1$. If the program goes outside this boundary, it will converge to the tamper-evident states $S2$ (See Fig. 1(c)). Hence, we define tamper-evident stabilizing programs as follows.

Definition 13 *(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 T

- 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 Fig. 1(c)). 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*.

Definition 13 defines tamper-evident stabilization. Next, in Definitions 14 and 15, we argue how it relates to (pure) tamper evidence and (pure) stabilization. The intuition behind a (pure) tamper-evident system is that if the system is perturbed beyond its legitimate states then any such perturbation must always be preserved. Such a systems treat any perturbations outside legitimate states as *tampering* and requires that its evidence should never be erased. If we use $S1$ to denote the legitimate states of the program then (pure) tamper evidence requires that if the system is perturbed beyond $S1$ then this would be preserved, say by eventually satisfying a predicate $S2$ (that is disjoint from $S1$) at all times. Moreover, this should occur even in the presence of adversary actions. Hence, we define a pure tamper-evident system as follows:

Definition 14 *(Pure tamper evidence).* Let program $p = \langle S_p, \delta_p\rangle$ be *k-tamper-evident stabilizing* with adversary adv for *invariants* $\langle S1, S2\rangle$. We say that program p is *pure tamper-evident* with adversary adv for *invariants* $\langle S1, S2\rangle$ iff

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

Observe that, in Definition 13, if we require $T = S1$ then this corresponds to the definition of (pure) tamper-evident system.

Additionally, the definition of (pure) stabilization [15] is that starting from an arbitrary state, the program recovers to legitimate states. Therefore, we define it as follows:

Definition 15 *(Pure stabilization).* Let program $p = \langle S_p, \delta_p\rangle$ be *k-tamper-evident stabilizing* with adversary adv for *invariants* $\langle S1, S2\rangle$. We say that program p is *pure stabilizing* with adversary adv for *invariants* $\langle S1, S2\rangle$ iff

- S_p converges to $S1$.

Observe that in Definition 13, if we require $T = S_p$ and $\neg T = S2 = \phi$ then this corresponds to the definition of (pure) stabilizing system. Therefore, based on Definitions 14 and 15, tamper-evident stabilization captures a range of systems from the ones that are *pure tamper-evident* to the ones 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 [5], 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 16 *(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 $p \cup adv$,
- $U \wedge T$ converges to $S1$ in p, and
- $U \wedge \neg T$ $\langle adv, k\rangle$-converges to $S2$ in p.

Observe that if U equals $true$ then the above definition is identical to that of Definition 13.

3.1. Token ring

In this section, we illustrate the definition of tamper-evident stabilization in the context of the well-known token ring program [15]. In a token ring program, a token is passed among a set of processes arranged in a ring, where the token acts as a permit for entry into the critical sections. We assume that, the program is subject to two types of perturbations: transient faults and tampering/adversary actions. If transient faults occur, the variables of the program will get corrupted arbitrarily and, consequently, the program will recover to its set of legitimate states $S1$. Nevertheless, the adversary can cause the system to fail by tampering a process. In this situation, we need the program to recover to the tamper-evident states $S2$ where there is no token in the system. Next, we describe the token ring program and represent that this program is tamper-evident stabilizing.

Description of the program. The program consists of N processes arranged in a ring. Each process j, $0 \le j \le N-1$, 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, $1 \le j \le N-1$, has the token iff processes j and $j-1$ have not failed and $x.j \ne x.(j-1)$. If process j, $1 \le j \le N-1$, has a token then it copies the value of $x.(j-1)$. 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:

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

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

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

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:

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

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

Proof 1. If T_{tr} is true then the program is essentially the same as the token ring program from [16] 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 actions (that may fail a process) cannot prevent the program from reaching $S2_{tr}$. □

3.2. Traffic controller

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 [11] where a quick recovery is provided to safe states and complete recovery to legitimate states can be obtained later (or with human intervention).

3.2.1. Description of the program

In this program, we have an intersection with two one-way roads [10]. 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.

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 $1-j$.)

$$TC1_j :: (sig_j = G) \wedge (sig_k = R) \longrightarrow sig_j = Y$$

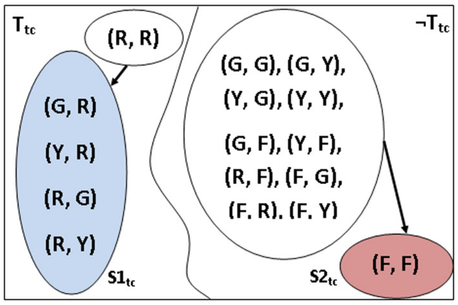


Fig. 2. Tamper-evident stabilizing traffic controller program.

$TC2_j :: (sig_j = Y) \wedge (sig_k = R) \longrightarrow (sig_j = R) \wedge (sig_k = G)$
$TC3_j :: (sig_j = R) \wedge (sig_k = R) \longrightarrow (sig_j = G)$
$TC4_j :: ((sig_j \neq R) \wedge (sig_k \neq R)) \vee (sig_k = F) \longrightarrow (sig_j = F)$
$TC5_j :: (sig_j = F) \wedge (sig_k = F) \longrightarrow$ *{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 2. If T_{tc} is true then the program is essentially the same as the traffic control program from [10] and, hence, it stabilizes to $S1_{tc}$. If the adversary TC_{adv} violates T_{tc}, the action $TC4_j$ 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 Fig. 2). □

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 3. 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 $\emptyset$ 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 *$\langle S, \phi\rangle$.*

Note that, if there exist 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 4. 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 $S1$.

5. Verification of tamper-evident stabilization

In this section, we show how to verify whether a given program p is tamper-evident stabilizing. This verification, which is represented in Algorithm 1, has three steps as follows.

1. **Finding predicate** T. First, we need to determine the predicate T (from Definition 13). Based on Definition 13, 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, the algorithm starts with the construction of T which is initialized by the invariant $S1$ (Lines 1–5). In each step of this construction, the algorithm adds a new state to T from which there is a transition to a state already in T. When there is no other state to add, the acceptability of T will be checked utilizing the following steps.
2. **Verifying whether** T **converges to** $S1$. To check the convergence to $S1$, based on Definition 6, first the algorithm checks the closure property of predicate T and invariant $S1$, and returns *false* if either of them is not closed (Lines 6–8 and 17–23). Then, it verifies if the program p has any cycle in $T - S1$ using the function *CheckCycle()* (Lines 9–11 and 24–30). The details of this function are as follows: it begins with the state space $Y = T - S1$ and transitions $\delta_Y = \{(s_0, s_1) \mid s_0 \in T - S1 \vee s_1 \in T - S1\}$. Since we have removed transitions that reach $S1$, Y may contain a deadlock state. If so, we remove that state from Y and the corresponding transitions that enter and exit that state. Upon termination, if any state 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 and the algorithm returns *false*.
3. **Verifying if** $\neg T$ **converges to** $S2$. Based on Definition 11, the algorithm needs to check the closure property of predicate $\neg T = S_p - T$ and invariant $S2$ in the presence of adversary adv (Lines 6–8 and 17–23). Moreover, it needs to check the cyclic-computations in $\neg T$ that do not include any state in $S2$ (Lines 13–15 and 24–30). For this purpose, utilizing the ideas in [12], we map the problem of verifying the convergence of $\neg T$ to the problem of verifying stabilization. Hence, first, we construct program p_1 from $\neg T$ (Line 12). Intuitively, program p_1 executes $k-1$ or more transitions of program p and then (optionally) one transition of the adversary. In other words, starting from an arbitrary state, program p continues its execution until the adversary executes. If adversary does not execute, p is guaranteed to converge; if there were a cycle involving a state, say s_1, then p_1 would involve a self-loop at s_1 preventing us from verifying convergence. Now, p is still in some state. To construct $p-1$, we need to partition subsequent execution by focusing on when/if adversary executes again. The resulting sequence is a transition of p_1. In constructing program p_1, $reach(s_0, s_1, l)$ denotes that s_1 can be reached from s_0 by execution of exactly l transitions of p. If such cycles of p_1 do not exist then p is tamper-evident stabilizing.

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

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

Proof 5. First, we show that there is at most one choice of T to demonstrate tamper-evident stabilization and this choice is computed in the loop on Lines 2–5. Let X denote the predicate constructed at the end of this loop. From every state in X, there is a path to $S1$. Given that $S1$ and $S2$ are disjoint this implies that from any state in X, there exists a computation of p that does not reach $S2$. Hence, if T is a choice to demonstrate tamper-evident stabilization then $\neg T$ and X must be disjoint. Also, by construction, there is no path from any state in $\neg X$ to a state in $S1$. It follows that there exists a computation of p that starts from $\neg X$ and does not reach $S1$. Therefore, $\neg X$ and T must be disjoint. Consequently, the only choice for T is predicate X constructed at the end of the loop on Lines 2–5.

The proofs of closure properties and construction of state predicate T can be trivially verified by considering each transition in δ_p. The *CheckCycle()* function can also be verified in polynomial-time in $|S_p|$ using the algorithm explained in Algorithm 1 (Lines 24–30). For constructing program p_1, we use the ideas from [12], which verifies a program in the presence of an adversary adv in polynomial-time in $|S_p|$. Hence, given a program p, state predicates $S1$ and $S2$, tamper-evident stabilization of program p for invariants $\langle S1, S2\rangle$ can be solved in polynomial time in $|S_p|$. □

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
3:     $T1 = T$;
4:     $T = T1 \cup \{s_0 \mid (s_0, s_1) \in p \ \wedge \ s_1 \in T\}$;
5: until $(T1 == T)$

6: if $\neg$(CheckClosure($T$, $p$) $\wedge$ CheckClosure($\neg T$, $p \cup adv$) $\wedge$ CheckClosure($S1$, $p$) $\wedge$
   CheckClosure($S2$, $p \cup adv$)) then
7:     return false;
8: end if

9: if CheckCycle($T - S1, p$) $\neq \emptyset$ then
10:     return false;
11: end if

12: $p_1 = \{(s_0, s_1) \mid (\exists l : l \geq k-1 : reach(s_0, s_1, l)) \ \vee \ (\exists s_2 : reach(s_0, s_2, l) \wedge (s_2, s_1) \in adv)\}$;

13: if CheckCycle($\neg T - S2, p_1$) $\neq \emptyset$ then
14:     return false;
15: end if

16: return true;

17: function CheckClosure(X, p)
18:     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
19:         return true;
20:     else
21:         return false;
22:     end if
23: end function

24: function CheckCycle($Y$, $p$)
25:     repeat
26:         $Y_1 = Y$;
27:         $Y = Y_1 - \{s_0 \mid \forall s_1 \in Y : (s_0, s_1) \notin \delta_p\}$;
28:     until $(Y_1 == Y)$
29:     return $Y$;
30: end function
```

6. Applications of tamper-evident stabilization

Tamper-evident stabilization has several potential applications. In this section, we briefly explain some of them.

6.1. Power grids

Our first application of tamper-evident stabilization is in the area of power grids. Power grids consist of several generating stations, transmission lines and consumption centers. From the perspective of the utility, we can view the power grid as a hierarchical system. Each level in the hierarchy has a certain level of consumption requirements, production capacity and ability to obtain electricity from other parts of the grid. Currently, in the power grids, you just expect electricity to flow when you plug a device in. However, with an emerging market for *smart loads* where the load will negotiate with the grid about energy consumption, electricity rates, etc, we envision a system where 50% of the load will be smart load in the future. In such systems, called *smart grids*, each subgrid will be associated with the available production capacity, smart load consumption requirements/production availability, ordinary load usage, etc.

With these conditions, intuitively, to apply tamper-evident stabilization in power grids, S_1 (Fig. 1(c)) would correspond to configurations where at each level in the hierarchy, the production capacity and consumption requirements are matched evenly. It would also provide sufficient flexibility in the subgrid for cases where the ordinary load increases/decreases upto a certain threshold. Predicate T would correspond to states where the grid is subject to faults or other environmental condition where ideal requirements cannot be satisfied. This may include reduced flexibility for increased load. Inside T, the subgrid is expected to resolve the situation automatically (with minimal operator intervention) by either adding increased supply, reducing the load or by obtaining electricity from other subgrids. Predicate $\neg T$ would correspond to the case where the grid has suffered a security attack where restoring it to full capacity (S_1) is not feasible. In this case, the system might terminate some of the smart load, cut off parts of the subgrid, etc. Predicate S_2 would correspond to the situation where the grid has restored itself to lower functionality based on the security attack. This restoration would be achieved with minimal/no operator intervention.

As an illustration of this idea with a simple example, consider an abstract version of the smart grid example described in [32]. In this example, a generator G serves two loads denoted by Z_1 and Z_2 (see Fig. 3). The system has three sensors: S_G,

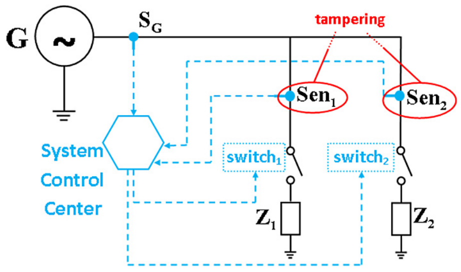


Fig. 3. Tamper-evident single generator smart grid system.

Sen_1, and Sen_2. Sensor S_G shows the power generated by the generator, and sensors Sen_1 and Sen_2 represent the demand of loads Z_1 and Z_2, respectively. The objective is to ensure that proper load shedding is utilized if the load is too high (respectively, generating capacity is too low). The system control center, represented by a hexagon in Fig. 3, communicates control signals to each of the two switches. These signals control the switches such that the following conditions are satisfied:

- Both switches are turned on if the overall sensed value does not exceed generation;
- If the sensed overall load demand exceeds generation, the system controller sheds one or both loads;
- If it appears that only one can be served, then the smaller load is shed assuming the larger load can be served by G; otherwise, the smaller load is served, and
- If neither load can individually be served by G then both loads are shed.

To model this system, we define the following variables:

- $V_j, (j = G, 1, 2)$: The value of sensors V_G, V_1 and V_2 respectively, and
- $switch_j, (j = 1, 2)$: The status of $switch_j$. If the switch is open, we show it by off and if it is close, we show it by on.

Based on the constraints explained and the variables described, we define the smart grid system by the following actions:

$$SG1 :: (V_1 + V_2 \leq V_G) \longrightarrow switch_1 = on \wedge switch_2 = on$$
$$SG2 :: (V_1 \leq V_G) \wedge (V_2 > V_G) \longrightarrow switch_1 = on \wedge switch_2 = off$$
$$SG3 :: (V_1 > V_G) \wedge (V_2 \leq V_G) \longrightarrow switch_1 = off \wedge switch_2 = on$$
$$SG4 :: (V_1 > V_G) \wedge (V_2 > V_G) \longrightarrow switch_1 = off \wedge switch_2 = off$$
$$SG5 :: ((V_1 + V_2 > V_G) \wedge (V_1 \leq V_G) \wedge (V_2 \leq V_G) \wedge (V_1 \leq V_2)) \longrightarrow switch_1 = off \wedge switch_2 = on$$
$$SG6 :: ((V_1 + V_2 > V_G) \wedge (V_1 \leq V_G) \wedge (V_2 \leq V_G) \wedge (V_1 > V_2)) \longrightarrow switch_1 = on \wedge switch_2 = off$$

Adversary actions. As a typical cyber attack, the adversary SG_{adv} can tamper with the sensor information, so the load management involves incorrect decision-making. We assume that the adversary can only change the sensor values Sen_1 and Sen_2 to greater values. Hence, the adversary actions can be represented as $(j = 1, 2)$:

$$SG_{adv_1} :: V_1 \leq V_G \longrightarrow V_1 = X, \text{ where } X > V_G$$
$$SG_{adv_2} :: V_2 \leq V_G \longrightarrow V_2 = Y, \text{ where } Y > V_G$$

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

$$T_{sg} = V_1 + V_2 \leq V_G$$
$$S1_{sg} = (V_1 + V_2 \leq V_G) \wedge (switch_1 = on \wedge switch_2 = on)$$
$$S2_{sg} = (V_1 + V_2 > V_G) \wedge (switch_1 = off \vee switch_2 = off)$$

6.2. Secure chip design

Another important application for tamper-evident stabilization is in the area of secure chip design. While chip designers utilize formal methods to verify that the chip does what it is supposed to do, they do not typically address *what else does the chip do?*. In particular, can the chip be used to do some unexpected things by providing invalid inputs or undesired environment. Designing a chip to be stabilizing may be desirable but likely to be impossible. By contrast, designing a chip to be tamper-evident stabilizing is easier since it allows the designer to identify a subset of the state space as unrecoverable and thereby allows it to simplify the recovery process (from recoverable states) and reaching tamper-evident states (from unrecoverable states).

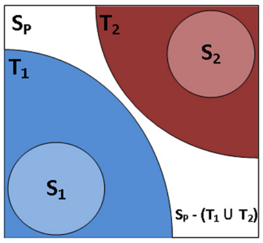


Fig. 4. Structure of a flexible tamper-evident stabilizing system.

6.3. *Intrusion detection*

Tamper-evident stabilization is also beneficial in intrusion detection and recovery. Typical intrusion detection systems prevent certain levels of intrusions outright, detect (but may be unable to prevent) a slightly higher level of intrusions, and potentially shut down the system in the event where tolerating intrusions is impossible. Tamper-evident stabilization can assist in the design of such systems to ensure that the system either recovers from intrusions or it reaches a tamper evident state.

This example also helps identify the question of what happens if the program ever reaches a tamper-evident state. Specifically, consider a system that is tamper-evident stabilizing with adversary adv for $\langle S1, S2\rangle$. Now, consider the question of what happens after the system reaches a state in $S2$. The purpose of $S2$ is to demonstrate evidence of tampering. In the context of above example, it is expected that if the system ever reaches a state in $S2$ then it will eventually be restored (e.g., with backup data, better firewall, improved intrusion detection system, etc.) to $S1$. However, actions that are required to perform this are not part of the original program. These actions are performed with a different program (humans?) under a different setting (controlled setting with assumptions about having no faults, adversaries etc.).

This analysis also helps in the use of tamper-evident stabilization in providing a tradeoff between fault-prevention techniques and fault-tolerant techniques. In broad terms, fault-prevention techniques utilize extra work (via space redundancy, time redundancy) to ensure that faults never happen or are not visible beyond a level of abstraction. By contrast, fault-tolerant techniques do not prevent occurrence of faults but utilize a lower level of redundancy. Instead, they focus on recovery when faults do happen. Tamper-evident stabilization potentially provides a tradeoff between these approaches. Specifically, we could use fault-tolerant mechanism to ensure that in any environment, the system will recover to either $S1$ or $S2$. If it recovers to $S2$ then fault-prevention techniques could be deployed, at increased cost, to restore the system to $S1$. Such tradeoff between fault-prevention and fault tolerance would be especially useful in reconfigurable systems (e.g., FPGAs) where certain components can be reconfigured to perform different activities at different times.

7. Flexible tamper-evident stabilization

As stated in Definition 13, the state space in tamper-evident stabilizing programs is divided into two parts T and $\neg T$, and the boundary between T and $\neg T$ is fixed. In some cases, this boundary is not enough and some states outside T and $\neg T$ may be reached, where we may not be able to distinguish how they are reached and how to proceed. Therefore, instead of having only T and $\neg T$, we need to divide the state space into three parts: T_1, T_2, and $\neg(T_1 \vee T_2)$ (see Fig. 4). The operations of the system in T_1 and T_2 would remain the same as those in T and $\neg T$ respectively. However, if the system reaches a state in $\neg(T_1 \vee T_2)$, which is a boundary in between, a human intervention is necessary to decide if the system needs to converge to $S1$ or $S2$. Based on this intuition, in this section, we define the notion of *flexible tamper-evident stabilization* (Section 7.1), compare them to tamper-evident stabilizing systems (Section 7.2), and propose an algorithm that verifies if a given program is flexible tamper-evident stabilizing (Section 7.3).

7.1. *Definition of flexible tamper-evident stabilization*

To define the notion of flexible tamper-evident stabilization, we utilize two predicates T_1 and T_2. If the program is in T_1, it will converge to legitimate states $S1$. If the program is in T_2, it will converge to tamper-evident states $S2$. Nevertheless, if it is in $\neg(T_1 \vee T_2)$, it has both options of converging to $S1$ and converging to $S2$. In this situation, an operator will choose which invariant the program needs to converge to.

The intuition behind flexible tamper-evident stabilization is that it is often impossible to distinguish the *cause* of reaching a given state from *inside the system*. And, it is possible to have the same state reached by the occurrence of an adversary action as well as by (random) faults. As an illustration, consider the power grid example in Section 6. Suppose that in this application, the voltage value can be perturbed by normal behavior as well as by adversary behavior. Suppose that in 99% of behaviors the fluctuation can only be 5%. However, under some rare cases (say 1%), the perturbation could be upto 10%. Furthermore, the adversary could also attempt to perturb it. Now, if the perturbation is more than 10%, it is certain that it was caused by an adversary. If it was less than 5%, we could treat it as normal perturbation and attempt to provide recovery. However, in the middle region, the operator will be provided a choice of both options.

Another example for flexible tamper-evident stabilization arises when assumptions made about system behavior are violated. For instance, multiple nodes in a distributed system may simultaneously fail due to adversary actions or by rare events (e.g., natural disasters that cause massive failures). In this case, external knowledge (e.g., existence of natural disasters in the area) could be used by the operator to identify whether the system should recover to its normal behavior. Hence, the definition of flexible tamper-evident stabilizing program is as follows.

Definition 17 *(Flexible 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 *flexible k-tamper-evident stabilizing* with adversary adv for *invariants* $\langle S1, S2\rangle$ iff there exist state predicates T_1 and T_2 such that:

- T_1 converges to S_1 in p,
- T_2 $\langle adv, k\rangle$-converges to S_2 in p, and
- for every state in $\neg(T_1 \vee T_2)$:
  - there exists a computation of p that reaches a state in T_1, **and**
  - there exists a computation of p that reaches a state in T_2.

From the above definition, it follows that if $\neg(T_1 \vee T_2)$ is empty, flexible tamper-evident stabilization would be similar to tamper-evident stabilization. In addition, if $\neg(T_1 \vee T_2)$ contains a state then there exists a computation that starts from that state and reaches T_1 and there also exists a computation that starts from that state and reaches T_2.

We extend the notion of flexible tamper-evident stabilization by simply overloading its definition with an environment.

Definition 18 *(Flexible 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 *flexible k-tamper-evident stabilizing* with adversary adv for *invariants* $\langle S1, S2\rangle$ in environment U iff there exist state predicates T_1 and T_2 such that:

- $S1, S2, T_1$, and T_2 are subsets of U,
- U is closed in $p \cup adv$,
- $U \wedge T_1$ converges to $S1$ in p,
- $U \wedge T_2$ $\langle adv, k\rangle$-converges to $S2$ in p, and
- $U \wedge \neg(T_1 \vee T_2)$: for every state in $\neg(T_1 \vee T_2)$:
  - there exists a computation that reaches a behavior in T_1, and
  - there exists a computation that reaches a behavior in T_2.

Observe that if U equals $true$ then the above definition is identical to that of Definition 17.

7.2. Flexible tamper-evident stabilization and tamper-evident stabilization

In this section, we compare the notion of flexible tamper-evident stabilization and tamper-evident stabilization. As mentioned in Section 7.1, if $\neg(T_1 \vee T_2)$ is empty, flexible tamper-evident stabilization would be equal to tamper-evident stabilization. Hence, every tamper-evident stabilizing program is also flexible tamper-evident stabilizing but not vice versa, which shows that tamper-evident stabilization is strictly weaker. We represent this claim by the following theorem.

Theorem 6. *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* flexible k-tamper-evident stabilizing *with adversary adv for* invariants $\langle S1, S2\rangle$.

Proof 6. To prove flexible tamper-evident stabilization, we need to identify values of T_1 and T_2. We set $T_1 = T$ and $T_2 = \neg T$. Therefore, T_1 converges to $S1$ in p and T_2 $\langle adv, k\rangle$-converges to $S2$ in p. Also, $\neg(T_1 \vee T_2)$ is empty since $T \vee \neg T = true$. This completes the proof. □

Nevertheless, if a program p is *flexible k-tamper-evident stabilizing* with adversary adv for *invariants* $\langle S1, S2\rangle$, we cannot guarantee that this program is *k-tamper-evident stabilizing* with adversary adv for *invariant* $\langle S1, S2\rangle$. This is due to the fact that if $\neg(T_1 \vee T_2)$ is not empty, there exists a state that has transitions to both T_1 and T_2, thereby violating the conditions of the program p to be *k-tamper-evident stabilizing*.

As an illustration, consider program $p = \langle S_p, \delta_p\rangle$ in Fig. 5, where $S_p = \{st_1, st_2, st_3\}$ and $\delta_p = \{(st_3, st_1), (st_3, st_2)\}$. This program is a flexible tamper-evident stabilizing program since $T_1 = S1 = st_1$, $T_2 = S2 = st_2$, and $\neg(T_1 \vee T_2) = st_3$. Nonetheless, it is not a tamper-evident stabilizing program since st_3 cannot be in either T or $\neg T$. It cannot be in T since every path in T reaches $S1$ while st_3 has paths to both $S1$ and $S2$. Similarly, st_3 is not in $\neg T$ since it has a path to $S1$. Therefore, we have a state that is not in either T or $\neg T$.

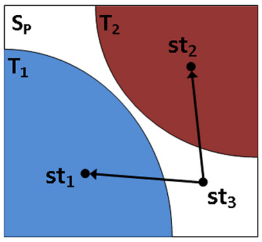


Fig. 5. An illustrative example for flexible tamper-evident stabilizing systems.

Algorithm 2 Verification of flexible tamper-evident stabilization.

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

```
1: $X_1 = S1$;
2: $X_2 = S2$;
3: repeat
4:     $temp_1 = X_1$;
5:     $temp_2 = X_2$;
6:     $X_1 = temp_1 \cup \{s_0 \mid (s_0, s_1) \in p \ \wedge \ s_1 \in X_1\}$;
7:     $X_2 = temp_2 \cup \{s_0 \mid (s_0, s_1) \in p \ \wedge \ s_1 \in X_2\}$;
8: until $(X_1 == temp_1 \wedge X_2 == temp_2)$

9: $FlexibleZone$ = $X_1 \cap X_2$;
10: $T_1 = X_1 - FlexibleZone$;
11: $T_2 = X_2 - FlexibleZone$;

12: if $X_1 \cup X_2 \neq S_p$ then
13:     return $false$;
14: end if

15: if $\neg$(CheckClosure($T_1$, $p$) $\wedge$ CheckClosure($T_2$, $p \cup adv$) $\wedge$ CheckClosure($S1$, $p$) $\wedge$
    CheckClosure($S2$, $p \cup adv$)) then
16:     return $false$;
17: end if

18: if CheckCycle($T_1 - S1$, $p$) $\neq \emptyset$ then
19:     return $false$;
20: end if

21: $p_1 = \{(s_0, s_1) \mid (\exists l : l \geq k-1 : reach(s_0, s_1, l)) \ \vee \ (\exists s_2 : reach(s_0, s_2, l) \ \wedge \ (s_2, s_1) \in adv)\}$

22: if CheckCycle($T_2 - S2$, $p_1$) $\neq \emptyset$ then
23:     return $false$;
24: end if

25: return $true$;

26: function CheckClosure(X, p)
27:     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
28:         return $true$;
29:     else
30:         return $false$;
31:     end if
32: end function

33: function CheckCycle($Y$, $p$)
34:     repeat
35:         $Y_1 = Y$;
36:         $Y = Y_1 - \{s_0 \mid \forall s_1 \in Y : (s_0, s_1) \notin \delta_p\}$;
37:     until $(Y_1 == Y)$
38:     return $Y$;
39: end function
```

7.3. Verification of flexible tamper-evident stabilization

Proving that a given program p is flexible tamper-evident stabilization is shown in Algorithm 2, and contains four steps as follows.

1. **Finding predicates T_1 and T_2, and $\neg(T_1 \vee T_2)$.** Based on Definition 17, from every state in T_2, we must eventually reach a state in $S2$. Hence, from T_2, we cannot reach a state in $S1$ or $\neg(T_1 \vee T_2)$. Also, from every state in T_1, we must

reach a state in $S1$. Thus, the only possible choice for T_1 is the states from where the program can reach $S1$. Therefore, we start from $S1$ and $S2$ to construct T_1 and T_2 respectively (Lines 1–11). When there is no other state to add to X_1 and X_2, we need to find the intersection of X_1 and X_2 since there might be some states in program p that have paths to both $S1$ and $S2$ (Line 9). To construct T_1 and T_2, we remove these states from X_1 and X_2 and add them to the *FlexibleZone* (Lines 10–11).

2. **Verifying if T_1 converges to** $S1$. To verify the convergence of T_1 to $S1$, first we check the closure property of T_1 and $S1$ (Lines 15–17 and 26–32). Then we verify if $T_1 - S1$ contains any cycle (Lines 18–20 and 33–39) and returns *false* if it does.
3. **Verifying whether T_2 converges to** $S2$. Similar to that in Algorithm 1, to verify the convergence of T_2 to $S2$, first we check the closure property of T_2 and $S2$ (Lines 15–17 and 26–32), and then we construct program p_1 (Line 21) and verify if $T_2 - S2$ contains any cycle in p_1 (Lines 22–24). The algorithm returns *false* if any cycle is found.
4. **Verifying if $\neg(T_1 \vee T_2)$ can reach both** $S1$ **and** $S2$. From Lines 12–14, if we return *true* then $X_1 \vee X_2$ is true. Hence, along with the definitions of T_1 and T_2, $\neg(T_1 \vee T_2)$ is the same as FlexibleZone on Line 9. By construction, there is a path from every state in FlexibleZone to a state in S_1 and to a state in S_2.

Theorem 7. *The following problem can be solved in polynomial time in* $|S_p|$

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

Proof 7. The choice of T_1 and T_2 (Lines 10 and 11), is the only possible choice to provide flexible tamper-evident stabilization. Based on the properties of flexible tamper-evident stabilization, from any state in T_1 there must be a path to $S1$ and there must not be a path to $S2$. By construction of X_1 and X_2 (Lines 1–8), X_1 is the set of states from where there is a path to S_1 and X_2 is the set of states from where there is a path to S_2. Hence, T_1 must be a subset of $X_1 - (X_1 \cap X_2)$. Furthermore, let s be a state in $X_1 - (X_1 \cap X_2)$. By construction, there is no path from s to S_2. Therefore, s cannot be in T_2. Likewise, it cannot be in $\neg(T_1 \vee T_2)$. It follows that, to verify flexible tamper-evident stabilization of program p, the choice of T_1 must be that in Line 10. Similarly, the choice of T_2 must be that in Line 11. This also shows that there is at most one choice of $\neg(T_1 \vee T_2)$. Additionally, following Theorem 5, CheckClosure and CheckCycle functions can be verified in polynomial-time in $|S_p|$. Hence, given a program p, state predicates $S1$ and $S2$, flexible tamper-evident stabilization of program p for invariants $\langle S1, S2\rangle$ can be solved in polynomial time in $|S_p|$. $\square$

8. Composition of tamper-evident stabilization and flexible tamper-evident stabilization

In this section, we evaluate the composition of tamper-evident stabilizing (Section 8.1) and flexible tamper-evident stabilizing (Section 8.2) systems by investigating different types of compositions considered for stabilizing systems.

8.1. Composing tamper-evident stabilization

This section evaluates the synchronous parallel compositions, asynchronous parallel composition, and transitivity of tamper-evident stabilizing systems.

8.1.1. Synchronous parallel composition

A synchronous parallel composition of two programs considers the case where two independent programs are run in parallel so that each program executes its actions in each transition. Therefore, in synchronous parallel composition, 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 synchronous parallel composition of p and q (denoted by $p\ []_s\ q$) also tamper-evident stabilizing?

Theorem 8 *(Synchronous parallel composition 1). 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 $\langle S1', S2'\rangle$
$\Rightarrow$
$p\ []_s\ q$ is tamper-evident stabilizing with adversary adv for $\langle S1 \wedge S1', S2 \vee S2'\rangle$.

Proof 8. Let T and T' be the state predicates utilized to demonstrate tamper-evident stabilization of p and q respectively. Then, the predicate used to demonstrate tamper-evident stabilization of $p\ []_s\ q$ is $T \wedge T'$. If $p\ []_s\ q$ begins in a state in $T \wedge T'$ then by tamper-evident stabilization property of p and q, each is guaranteed to recover to $S1 \wedge S1'$. And, if $p\ []_s\ q$ begins in a state where $T \wedge T'$ is false then by the tamper-evident stabilization property of p and q, either p will reach a state in $S2$ or q will reach a state in $S2'$. □

Theorem 9 *(Synchronous parallel composition 2). 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 $\langle S1', S2'\rangle$
$\Rightarrow$
$p\ []_s\ q$ is tamper-evident stabilizing with adversary adv for $\langle S1 \vee S1', S2 \wedge S2'\rangle$.

Proof 9. Let T and T' be the state predicates utilized to demonstrate tamper-evident stabilization of p and q respectively. Also, assume the predicate used to demonstrate tamper-evident stabilization of $(p\ []_s\ q)$ is $(T \vee T')$. If $(p\ []_s\ q)$ begins in a state in $T \vee T'$ then by tamper-evident stabilization property of p and q, it is guaranteed that either p will reach a state in $S1$ or q will reach a state in $S1'$. And, if $p\ []_s\ q$ begins in a state where $T \vee T'$ is false (or $\neg T \wedge \neg T'$ is true) then by the tamper-evident stabilization property of p and q, both p and q will reach a state in $S2 \wedge S2'$. □

Observe that in Theorem 8, predicate T is instantiated to be $T \wedge T'$. In other words, in this case, both programs are perturbed to an acceptable level. This theorem would be especially useful when combining two programs that collaborate with each other. Hence, it is expected that as long as neither is perturbed beyond an acceptable level, both would recover to their legitimate states. By contrast, in Theorem 9, predicate T is instantiated to be $T \vee T'$. In other words, at least one of the programs is not perturbed beyond an acceptable level. This theorem would be especially useful when the components are used for the purpose of redundancy. Hence, as long as one of the components is not perturbed beyond an acceptable level, it would recover to a legitimate state.

8.1.2. Asynchronous parallel composition

In an asynchronous parallel composition of two independent programs, the programs are run in parallel on a *weakly fair scheduler*. A weakly fair scheduler in our context has the following properties. If composing programs p and q, in every step, the scheduler allows either p or q to execute. Furthermore, if program p (respectively q) is in a state where some transition of it can be executed then p (respectively q) is allowed to execute in some step. In other words, a weakly fair scheduler does not allow only one program to execute forever even when the other program could execute. To discuss tamper-evident stabilization of asynchronous parallel composition of two tamper-evident stabilizing programs, let us first define *atomic* tamper-evident stabilizing programs as follows:

Atomic tamper-evident stabilization. A program p is atomic tamper-evident stabilizing iff:

- It is tamper-evident stabilizing, and
- $\forall (s_0, s_1) \in p : (s_0 \in \neg T) \Rightarrow (s_1 \in S2)$, where T is the predicate defined in Definition 13.

Now, consider two programs p and q that are atomic tamper-evident stabilizing for $\langle S1, S2\rangle$ and $\langle S1', S2'\rangle$ respectively, and do not share any variables. Next, we explain if the asynchronous parallel composition of p and q (denoted by $p\ []_a\ q$) is also tamper-evident stabilizing.

Theorem 10 *(Asynchronous parallel composition). Given programs p and q that do not share variables.*

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

Proof 10. Let T and T' be the state predicates utilized to demonstrate atomic tamper-evident stabilization of p and q respectively. Then, the predicate used to demonstrate atomic tamper-evident stabilization of $p\ []_a\ q$ is $T \wedge T'$. If $p\ []_a\ q$ begins in a state in $T \wedge T'$ then by tamper-evident stabilization property of p and q, each is guaranteed to recover to $S1 \wedge S1'$. Moreover, if $p\ []_a\ q$ begins in a state where $T \wedge T'$ is false then either $\neg T$ or $\neg T'$ is true. In an arbitrary interleaving, one of the programs executes the first action. Let p be the one that executes first. Since p is atomic tamper-evident stabilizing, p reaches a state in $S2$ in an atomic step. Subsequently, $\neg T'$ $\langle adv, k\rangle$-converges to $S2'$ in q. □

Observation 1. Consider that if programs p and q are not atomic tamper-evident stabilizing, their asynchronous parallel composition cannot be guaranteed. Since p and q are tamper-evident stabilizing, p *$\langle adv, k\rangle$-converges* to $S2$ and q *$\langle adv, k\rangle$-converges* to $S2'$. However, the asynchronous composition of p and q in the presence of adv may create computations where the actions of p and q are interleaved and p (respectively, q) is not given the chance of taking at least

$k-1$ actions between any two adversary actions. As such, there are no guarantees that p (respectively, q) will recover to $S2$ (respectively, $S2'$).

Observation 2. If we can ensure that the adversary stops for a long enough amount of time so that at least one of the programs p or q can recover to $S2$ or $S2'$, the atomicity of p and q is not necessary in Theorem 10. In other words, the number of steps p or q needs to recover to $S2$ or $S2'$ has to be smaller than k in Definition 13.

Observation 3. In asynchronous 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 $p\ []_a\ q$ 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, $S2'$).

8.1.3. Transitivity

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

Theorem 11 *(Transitivity 1). Given program p.*

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* $\langle S1' \vee S2', S2 \rangle$ *in* U.

Proof 11. Let T and T' be predicates used to demonstrate tamper-evident stabilization of p in the first and second antecedents respectively. The predicate used to demonstrate tamper-evident stabilization of p in U is T. If we begin in a state in $U \wedge T$ then p is guaranteed to recover to $S1$ (by first antecedent) and recover to $(S1' \vee S2')$ (by second antecedent). And, if it begins in a state in $U \wedge \neg T$ then it is guaranteed to reach $S2$ even if it is perturbed by the adversary (by first antecedent). Hence, the proof follows. □

We can also infer transitivity property by the following theorem.

Theorem 12 *(Transitivity 2). Given program p.*

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* $\langle S1', S2' \rangle$.

Proof 12. The proof is similar to that of Theorem 11. □

8.2. Composing flexible tamper-evident stabilization

This section evaluates whether the theorems presented in Sections 8.1 are also true for flexible tamper-evident stabilizing systems.

8.2.1. Synchronous parallel composition

Assume that the definition of synchronous parallel composition of two programs p and q is the same as that in Section 8.1. Hence, the synchronous parallel composition of two flexible tamper-evident stabilizing programs is as follows.

Theorem 13 *(Synchronous parallel composition 1). Given programs p and q that do not share variables.*

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

Proof 13. Let $\{T_1, T_2\}$ be the state predicates utilized to demonstrate flexible tamper-evident stabilization of p. Also, the predicates used to demonstrate flexible tamper-evident stabilization of $p\ []_s\ q$ are $T_1'' = T_1$ and $T_2'' = T_2$. If $p\ []_s\ q$ begins in a state in T_1'' then by flexible tamper-evident stabilization property of p, it is guaranteed to recover to $S1$. Similarly, if $p\ []_s\ q$ begins in a state in T_2'' then the composition is guaranteed to reach a state in $S2$. Moreover, if it begins in a state in $\neg(T_1'' \vee T_2'')$, we have both computations to reach a state in either $S1$ and $S2$. This completes the proof. □

Corollary 2. *The same theorem holds for* $\langle S1', S2' \rangle$.

Theorem 14 *(Synchronous parallel composition 2). Given programs p and q that do not share variables.*

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

Proof 14. Let $\{T_1, T_2\}$ and $\{T_1', T_2'\}$ be the state predicates utilized to demonstrate flexible tamper-evident stabilization of p and q respectively. Then, the predicates used to demonstrate flexible tamper-evident stabilization of $p\ []_s\ q$ are $T_1'' = T_1 \wedge T_1'$ and $T_2'' = T_2 \wedge T_2'$. If $p\ []_s\ q$ begins in a state in T_1'' then by flexible tamper-evident stabilization property of p and q, each is guaranteed to recover to $S1 \wedge S1'$. Similarly, if $p\ []_s\ q$ begins in a state in T_2'' then the composition is guaranteed to reach a state in $S2 \wedge S2'$. Moreover, if it begins in a state in $\neg(T_1'' \vee T_2'')$, we have: $\neg(T_1'' \vee T_2'') = \neg(T_1 \wedge T_1') \wedge \neg(T_2 \wedge T_2') = (\neg T_1 \vee \neg T_1') \wedge (\neg T_2 \vee \neg T_2')$. The first part, $(\neg T_1 \vee \neg T_1')$, illustrates that there exist computations that reach $S2 \vee S2'$, and the second part, $(\neg T_2 \vee \neg T_2')$, shows that there exist computations that reach states in $S1 \vee S1'$. Therefore, there exist computations that reach states in $S1 \vee S1'$ and $S2 \vee S2'$. □

8.2.2. Asynchronous parallel composition

In order to discuss flexible tamper-evident stabilization of asynchronous parallel composition of two flexible tamper-evident stabilizing programs, let us first define *atomic* flexible tamper-evident stabilizing programs as follows:

Atomic flexible tamper-evident stabilization. A program p is atomic flexible tamper-evident stabilizing iff:

- It is flexible tamper-evident stabilizing,
- $\forall (s_0, s_1) \in p : (s_0 \in T_2) \Rightarrow (s_1 \in S2)$, where T_2 is the predicate defined in Definition 17, and
- $\forall (s_0, s_1) \in p : (s_0 \in \neg(T_1 \vee T_2)) \Rightarrow (s_1 \in (S1 \vee S2))$, where T_1 and T_2 are the predicates defined in Definition 17.

Now, consider two programs p and q that are atomic flexible tamper-evident stabilizing for $\langle S1, S2 \rangle$ and $\langle S1', S2' \rangle$ respectively, and do not share any variables. Next, we explain if the asynchronous parallel composition of p and q (denoted by $p\ []_a\ q$) is also tamper-evident stabilizing.

Theorem 15 *(Asynchronous parallel composition). Given programs p and q that do not share variables.*

p is atomic flexible tamper-evident stabilizing with adversary adv for $\langle S1, S2 \rangle \wedge$
q is atomic flexible tamper-evident stabilizing with adversary adv for $\langle S1', S2' \rangle$
$\Rightarrow$
$p\ []_a\ q$ *is flexible tamper-evident stabilizing with adversary adv for*
$\langle S1 \vee S1', S2 \vee S2' \rangle$ *under weak fairness.*

Proof 15. The proof is similar to that of Theorem 13. □

Observation 4. Similar to Observation 1, if the programs p and q are not atomic flexible tamper-evident stabilizing, the asynchronous parallel composition of p and q cannot be guaranteed to be flexible tamper-evident stabilizing.

Remark 4. Theorem 15 cannot be true for $\langle S1 \wedge S1', S2 \vee S2' \rangle$ since if one program is perturbed to the boundary area then $\langle S1 \wedge S1' \rangle$ cannot be guaranteed.

8.2.3. Transitivity

In this section, we discuss if flexible tamper-evident stabilization preserves transitivity in a manner similar to stabilizing programs.

Theorem 16 *(Transitivity 1). Given program p.*

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

Proof 16. Let $\{T_1, T_2\}$ be the predicates used to demonstrate flexible tamper-evident stabilization of p in the first antecedent and $\{T'\}$ be the predicate used to demonstrate tamper-evident stabilization of p in the second antecedent. The predicates used to demonstrate flexible tamper-evident stabilization of p in U are T_1 and T_2. If we begin in a state in $U \wedge T_1$ then p is guaranteed to recover to $S1$ (by first antecedent) and recover to $(S1' \vee S2')$ (by second antecedent). Also, if it begins in a state in $U \wedge T_2$ then it is guaranteed to reach $S2$ even if it is perturbed by the adversary (by first antecedent). Furthermore, if we begin in a state in $\neg(T_1 \vee T_2)$, it will reach a state in either $S1$ or $S2$. □

Theorem 17 *(Transitivity 2). Given program p.*

p is flexible 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 flexible tamper-evident stabilizing with adversary adv for $\langle S1', S2'\rangle$.

Proof 17. The proof is similar to that of Theorem 16. □

9. Methodology for designing tamper-evident stabilization

In this section, we identify some possible approaches for designing tamper-evident stabilization. Tamper-evident stabilization contains two main requirements: (1) recovery to legitimate states in the presence of normal perturbations that the system is expected to tolerate, and (2) reaching tamper-evident states even if perturbed by an adversary. We evaluate the use of some of the existing approaches for designing stabilization in designing tamper-evident stabilization.

9.1. Local detection and global correction

One approach for designing stabilization is via local detection and global correction [7,38]. In this work, the program consists of a set of processes arranged in some connected topology. Specifically, each process is associated with a set of neighbors that it can communicate with. It reads the state of its neighbors and updates its own state. Furthermore, 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 [30]) to restore the system to a legitimate state.

A similar approach is also applicable for tamper-evident stabilization. Specifically, to achieve this, we view the program to consist of a set of processes/components such that 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 suppose $\forall j :: (S1.j \Rightarrow T.j) \wedge (S2.j \Rightarrow \neg T.j) \wedge \neg(S1.j \wedge S2.j)$. We also require that if $T.j$ is false then adversary actions cannot make it true. This captures that adversary cannot cause a process to *forget* that it was outside permissible states. Likewise, $S2.j$ is also closed in the adversary actions, i.e., once the process has reached a state that identifies that it has been tampered with, it cannot be undone.

In this case, the outline of the program for process j to obtain tamper-evident stabilization is as follows: Process j checks the values of $S1.j$, $S2.j$ and $T.j$ and takes actions as described by the following table.

S1.j	S2.j	T.j	Action
False	False	False	Satisfy $S2.j$
False	False	True	Initiate global correction to restore $S1$
False	True	False	No action
False	True	True	Impossible since $S2.j \Rightarrow \neg T.j$
True	False	False	Impossible since $S1.j \Rightarrow T.j$
True	False	True	No action (appears legitimate)
True	True	*	Impossible since $\neg(S1.j \wedge S2.j)$

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. Following the approach in [7,38], 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. This is due to the fact that convergence to $S1$ is required only from states in T (which is equal to $\forall j :: T.j$. Also, if $T.k$ is false at some process, we want to restore the system to a state in $S2$ (which is equal to $\forall j :: S2.j$.) This is achieved with the idea that if $T.j$ is false at some process then all processes (one by one) set $S2.j$ to be true. Since a process can only observe the state of its neighbors, if process j observes that $T.k$ is false for some neighbor k then j will satisfy $S2.j$. Since $S2.j \Rightarrow \neg T.j$, it follows that neighbors of j will set $S2.j$ to be true. Given the connected topology, this will guarantee that if $T.j$ is false for some process then the program will eventually reach a state in $S2$.

Effect of the structure of predicate T. In the above example, T was a conjunctive predicate whereas $\neg T$ was a disjunctive predicate. From T the system was required to converge to $S1$ (another conjunctive predicate). And, from $\neg T$ the program was required to converge to $S2$ (another conjunctive predicate). Thus, the natural question is could we do the same thing if $\neg T$ was a conjunctive predicate and T was a disjunctive predicate. Next, we argue that in this case, design of tamper-evident stabilization is expected to be more complicated.

To validate this claim, we consider a simple approach for designing tamper-evident stabilization for systems where T is a disjunctive predicate, i.e., $T = \exists j :: T.j$. In this system, if process j observes that $T.j$ is false, it cannot unilaterally decide to satisfy $S2.j$. This is because T may still be true even if $T.j$ is false. However, if j detects that $S1.j$ is false then it can initiate a global correction to restore $S1$. Now, consider the case where some process k observes that a global correction is being performed when $T.k$ is false. In this case, k cannot simply ignore that operation because the system may be in a state where T is still true. Moreover, it cannot participate since there is a possibility that T is false and, hence, the system should be restored to $S2$.

Intuitively, in this approach, we need a mechanism to *count* the number of processes that violate their T predicate. Under the assumption that an adversary cannot move the system from a state where $T.j$ is false to a state where $T.j$ is true, this could be achieved by maintaining some type of simple structure (tree, ring, etc) to count the number of processes that violate the T predicate. However, such an approach would suffer from faults that affect that structure and so on. By contrast, if T is a conjunctive predicate then any node can unilaterally detect if T is violated. We note that due to this reason, we chose T to be a conjunctive predicate in the above analysis.

9.2. Local detection and local correction

Another approach for stabilization is to utilize local detection and local correction. Intuitively, in this approach, the invariant S is of the form $\forall j :: S.j$. Moreover, the S predicates of different processes are arranged in a partial order. Actions that correct $S.j$ must preserve all predicates that come earlier in the order. However, they may violate predicates that come later in the order. In such a system, each process j is responsible for checking if $S.j$ is false. If it is then it is required to restore the system to a state where $S.j$ is true. This, in turn, may violate constraints that come later in the order. However, given that we have a partial order, eventually we reach a state where $S.j$ is true in all states.

We can use the same approach for designing tamper-evident stabilization. This approach is similar to that described above except that instead of initiating a global operation, each node simply corrects its own $S1$ predicate as needed. However, if $T.j$ is false for some node j then j does not perform its local correction. Instead, the node j changes its state to satisfy $S2.j$ and, thereby, cause the system to reach a state in $S2$.

9.3. Model repair and synthesis

An alternative method for the design of tamper-evident stabilization is based on exploiting previous work on using model repair or model synthesis of fault-tolerant programs [31,3,9,20] and self-stabilization in particular [1,2,21,22,28,29]. In such methods, one starts with a fault-intolerant (respectively, non-stabilizing) program and systematically explores the state space of the program towards designing convergence actions while preserving behavior in the legitimate states. The core of such a repair/synthesis method relies on designing convergence from T to $S1$ and convergence from $\neg T$ to $S2$. The definition of tamper-evident stabilization allows us to divide the problem of designing tamper-evident stabilization into two independent sub-problems since T and $\neg T$ are disjoint. This independence facilitates the use of parallel machines [18] in automated design of convergence of T to $S1$ and $\langle adv, k\rangle$-convergence of $\neg T$ to $S2$.

Methods for the design of convergence can be classified into two families: top-down and bottom-up. In the top-down method [20], the synthesis algorithms start with the weakest possible program that includes any transition that p can have in T. If after including all such transitions some deadlock states remain in T then convergence to $S1$ cannot be designed. Otherwise, the synthesis algorithms resolve non-progress cycles in $T - S1$ while preserving deadlock-freedom in $T - S1$. To resolve cycles, synthesis algorithms eliminate some convergence actions and check if an alternative combination of convergence actions would resolve the cycles without creating deadlocks.

In the bottom-up methods [21,18], synthesis algorithms start with a program that is cycle-free in $T - S1$ and incrementally include one convergence action at a time towards resolving deadlocks. If the included action resolves deadlocks but creates cycles with existing actions then the algorithms rollback and try a different convergence action. To perform this process more efficiently, some methods first design a ranking function that assigns a positive integer to every state in $T - S1$. This way $T - S1$ is partitioned to several layers. Such layers guide the synthesis algorithm to design convergence actions in a more systematic way. An example ranking function [21] is based on the length of the shortest computation from each state $s \in T - S1$ to some state in $S1$. As such, the first layer around $S1$ captures the states from where $S1$ can be reached in a single step, the second layer includes states from where $S1$ can be reached in two steps, and so forth.

While the design of convergence from T to $S1$ can directly benefit from existing work on algorithmic design, designing $\langle adv, k\rangle$-convergence from $\neg T$ to $S2$ introduces new challenges due to the requirement of $\langle adv, k\rangle$-convergence. One approach to tackle this problem is to ensure that, from every state $s \in \neg T - S2$, the program makes progress towards $S2$ between any two consecutive adversary actions. We can achieve this by ensuring that if we have a sequence of three states (s_0, s_1, s_2) such that (s_0, s_1) is a program transition and (s_1, s_2) is an adversary transition then rank of s_2 is not greater than the rank of s_0 (and s_1). This ensures that adversary might slow down the convergence to $S2$ but it cannot prevent it. A detailed algorithm for designing such recovery is beyond the scope of this paper. However, we note that such an algorithm can be designed based on the principles of convergence stair [25] and min–max computation. Finally, model repair and synthesis can also help in automating the previous two design methods mentioned in this section.

10. The relationship between tamper-evident stabilization and other stabilization techniques

Starting with Dijkstra's seminal work [15] 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., [23,39]) 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., [35,34]) programs tolerate the scenarios where a subset of processes is Byzantine. FTSS (e.g., [8]) covers stabilizing programs that tolerate permanent crash faults. Multitolerant stabilizing (e.g., [30,19]) systems ensure that, in addition to stabilization, the program masks a certain class of faults. Finally, active stabilization [12] 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., [24,14]) 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 [27] 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., [4,6]) 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 [13] 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., [33,36,37]). 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 stabilization 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 13) 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 [19] 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 13, 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 [19].

11. Conclusion and future work

The goal of this paper is to combine the notion of stabilization with tamper evidence. Intuitively, the notion of tamper evidence requires that any tampering of the system is preserved forever so that the user is aware of the tampering. By contrast, stabilization requires that the system recovers to legitimate states from arbitrary perturbation. In other words, it requires that any perturbation is eventually forgotten (when the system reaches a legitimate state). Although these two concepts appear contradictory, we showed that they can be effectively combined to define the concept of tamper-evident stabilization. Tamper-evident stabilization captures the intuition that *if the system is mildly perturbed then it is guaranteed to recover but if it is tampered that causes severe perturbation then the effect of that tampering is always preserved.*

We extended the notion of tamper-evident stabilization to flexible tamper-evident stabilization. The intuition of flexible tamper-evident stabilization was based on the observation that the *cause* of reaching a given state cannot always be identified *inside the system*. In other words, the same state could be reached by tampering or by some rare fault. In this case, it is desirable that the system provides an alternative to recover to a normal behavior as well as an alternative to recover to tamper-evident behavior. This allows the system operator to use *external factors* to guide the system into an appropriate state.

We formally defined tamper-evident and flexible tamper-evident stabilization and investigated how they relate 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*. Moreover, flexible tamper-evident stabilizing systems are a more general form of tamper-evident 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 (or flexible 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 program. Moreover, we find that parallel composition of tamper-evident or flexible tamper-evident stabilizing systems works in a manner similar to that of stabilizing systems. Nevertheless, transitivity requirements of tamper-evident stabilization and flexible tamper-evident stabilization are somewhat different than that for stabilizing systems.

Tamper-evident stabilization has important applications in chip design, dependable and secure protocols, and design of winning strategies in multi-player games. In particular, when it is impossible to recover from some states, tamper-evident stabilization provides an alternative to preserve evidence of tampering. Thus, it provides the designer an increased design space to design programs that are fault-tolerant and secure. We also presented power grid systems as another potential application for tamper-evident 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).

Acknowledgements

This work is supported by NSF CNS 1329807, NSF CNS 1318678, NSF CCF-1116546, and XPS 1533802. A preliminary version of this paper appears in FSEN 2015.

References

[1] Fuad Abujarad, Sandeep S. Kulkarni, Multicore constraint-based automated stabilization, in: 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems, 2009, pp. 47–61.

[2] Fuad Abujarad, Sandeep S. Kulkarni, Automated constraint-based addition of nonmasking and stabilizing fault-tolerance, Theor. Comput. Sci. 412 (33) (2011) 4228–4246.

[3] A. Arora, A Foundation of Fault-Tolerant Computing, PhD thesis, The University of Texas at Austin, 1992.

[4] A. Arora, M. Gouda, G. Varghese, Constraint satisfaction as a basis for designing nonmasking fault-tolerant systems, J. High Speed Netw. 5 (3) (1996) 293–306.

[5] A. Arora, M.G. Gouda, Closure and convergence: a foundation of fault-tolerant computing, IEEE Trans. Softw. Eng. 19 (11) (1993) 1015–1027.

[6] Anish Arora, Efficient reconfiguration of trees: a case study in methodical design of nonmasking fault-tolerant programs, in: Hans Langmaack, Willem P. de Roever, Jan Vytopil (Eds.), FTRTFT, in: Lect. Notes Comput. Sci., vol. 863, Springer, 1994, pp. 110–127.

[7] B. Awerbuch, B. Patt-Shamir, G. Varghese, Self-stabilization by local checking and correction, in: Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science, 1991, pp. 268–277.

[8] Joffroy Beauquier, Synnöve Kekkonen-Moneta, On ftss-solvable distributed problems, in: 3rd Workshop on Self-stabilizing Systems, WSS, 1997, pp. 64–79.

[9] B. Bonakdarpour, S.S. Kulkarni, Exploiting symbolic techniques in automated synthesis of distributed programs with large state space, in: Proceedings of the 27th International Conference on Distributed Computing Systems, June 2007, pp. 3–10.

[10] Borzoo Bonakdarpour, Sandeep S. Kulkarni, Compositional verification of fault-tolerant real-time programs, in: 9th ACM & IEEE International Conference on Embedded Software, EMSOFT, 2009, pp. 29–38.

[11] Borzoo Bonakdarpour, Sandeep S. Kulkarni, On the complexity of synthesizing relaxed and graceful bounded-time 2-phase recovery, in: 15th International Symposium on Formal Methods, FM, 2009, pp. 660–675.

[12] Borzoo Bonakdarpour, Sandeep S. Kulkarni, Active stabilization, in: 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems, SSS, 2011, pp. 77–91.

[13] J.E. Burns, M.G. Gouda, R.E. Miller, Stabilization and pseudo-stabilization, Distrib. Comput. 7 (1) (1993) 35–42.

[14] Stéphane Devismes, Sébastien Tixeuil, Masafumi Yamashita, Weak vs. self vs. probabilistic stabilization, in: 28th International Conference on Distributed Computing Systems, ICDCS, 2008, pp. 681–688.

[15] E.W. Dijkstra, Self-stabilizing systems in spite of distributed control, Commun. ACM 17 (11) (1974) 643–644.

[16] E.W. Dijkstra, A Discipline of Programming, Prentice-Hall, 1990.

[17] S. Dolev, Self-Stabilization, MIT Press, 2000.

[18] Ali Ebnenasir, Aly Farahat, Swarm synthesis of convergence for symmetric protocols, in: Proceedings of the Ninth European Dependable Computing Conference, 2012, pp. 13–24.

[19] Ali Ebnenasir, Sandeep S. Kulkarni, Feasibility of stepwise design of multitolerant programs, ACM Trans. Softw. Eng. Methodol. 21 (1) (December 2011) 1–49.

[20] Ali Ebnenasir, Sandeep S. Kulkarni, Anish Arora, FTSyn: a framework for automatic synthesis of fault-tolerance, Int. J. Softw. Tools Technol. Transf. 10 (5) (2008) 455–471.

[21] Aly Farahat, Ali Ebnenasir, A lightweight method for automated design of convergence in network protocols, ACM Trans. Auton. Adapt. Syst. 7 (4) (December 2012) 38:1–38:36.

[22] Aly Farahat, Ali Ebnenasir, Local reasoning for global convergence of parameterized rings, in: IEEE International Conference on Distributed Computing Systems, ICDCS, 2012, pp. 496–505.

[23] Sukumar Ghosh, Arobinda Gupta, An exercise in fault-containment: self-stabilizing leader election, Inf. Process. Lett. 59 (5) (1996) 281–288.

[24] M. Gouda, The theory of weak stabilization, in: 5th International Workshop on Self-Stabilizing Systems, in: Lect. Notes Comput. Sci., vol. 2194, 2001, pp. 114–123.

[25] M.G. Gouda, N. Multari, Stabilizing communication protocols, IEEE Trans. Comput. 40 (4) (1991) 448–458.

[26] G. Gouda Mohamed, Elements of security: closure, convergence, and protection, Inf. Process. Lett. 77 (2–4) (2001) 109–114, in honor of Edsger W. Dijkstra.

[27] Amos Israeli, Marc Jalfon, Token management schemes and random walks yield self-stabilizing mutual exclusion, in: Ninth Annual ACM Symposium on Principles of Distributed Computing, PODC, 1990, pp. 119–131.

[28] Alex Klinkhamer, Ali Ebnenasir, On the complexity of adding convergence, in: Proceedings of the 5th International Symposium on Fundamentals of Software Engineering, FSEN, 2013, pp. 17–33.

[29] Alex Klinkhamer, Ali Ebnenasir, On the hardness of adding nonmasking fault tolerance, IEEE Trans. Dependable Secure Comput. 12 (3) (2015) 338–350.

[30] S. Kulkarni, A. Arora, Multitolerance in distributed reset, Chic. J. Theor. Comput. Sci. 1998 (4) (December 1998).

[31] S.S. Kulkarni, A. Arora, Automating the addition of fault-tolerance, in: Formal Techniques in Real-Time and Fault-Tolerant Systems, 2000, pp. 82–93.

[32] Deepa Kundur, Xianyong Feng, Salman Mashayekh, Shan Liu, Takis Zourntos, Karen L. Butler-Purry, Towards modelling the impact of cyber attacks on a smart grid, IJSN 6 (1) (2011) 2–13.

[33] David Lie, Chandramohan A. Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John C. Mitchell, Mark Horowitz, Architectural support for copy and tamper resistant software, in: 9th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS, 2000, pp. 168–177.

[34] Mahyar R. Malekpour, A byzantine-fault tolerant self-stabilizing protocol for distributed clock synchronization systems, in: 8th International Symposium on Stabilization, Safety, and Security, SSS, 2006, pp. 411–427.

[35] Nesterenko Mikhail, Anish Arora, Tolerance to unbounded byzantine faults, in: 21st Symposium on Reliable Distributed Systems, SRDS, 2002, p. 22.

[36] Sean W. Smith, Steve Weingart, Building a high-performance, programmable secure coprocessor, Comput. Netw. 31 (8) (1999) 831–860.

[37] G. Edward Suh, Dwaine E. Clarke, Blaise Gassend, Marten van Dijk, Srinivas Devadas, AEGIS: architecture for tamper-evident and tamper-resistant processing, in: Proceedings of the 17th Annual International Conference on Supercomputing, ICS, 2003, pp. 160–171.

[38] G. Varghese, Self-Stabilization by Local Checking and Correction, PhD thesis, MIT, 1993.

[39] Hongwei Zhang, Anish Arora, Guaranteed fault containment and local stabilization in routing, Comput. Netw. 50 (18) (2006) 3585–3607.