On the (Im)possibility of Timed Tamper-Evident Software in (A)synchronous Systems

Jason Franklin and Michael Carl Tschantz
{jfrankli, mtschant}@cs.cmu.edu

Abstract

Tamper-evident software has the property that a verifier can detect a violation of program integrity during execution. In this paper, we study programs that through their own execution provide sufficient information in the form of responses and timing to detect tampering. We refer to such programs as *timed tamper-evident programs*.

We formalize the notation of a timed tamper-evident program, model a micro-controller under this formalization, and prove the existence of a timed tamper-evident program for this model when used with synchronous communication. We develop timed tamper-evident programs which verify both control and data integrity. We discuss the existence of timed tamper-evident functions in an asynchronous system.

1 Introduction

In the hostile host problem, a program executes on a host whose software is under the complete control of an active adversary who may attempt to tamper with program execution. To defend against this threat, a verifier must check the integrity of the program and the execution environment to guarantee that they are not modified either before or during program execution.

One possible solution to verify the integrity of the program and execution environment is to provide the verifier with direct low-level access to the host's hardware including the memory and register contents. Detailed inspection at a low-level is possible through the addition of specialized hardware directly on the host. Solutions to the hostile host problem based on tamper-resistant hardware and secure co-processors are a well studied and active area of research [11, 3, 1].

Rather than rely on low level access and specialized hardware, we study the existence of software solutions to the hostile host problem. Specifically, we evaluate the existence of software which through its execution provides a guarantee of its integrity and the integrity of the execution environment. We study the following questions:

1. Does tamper-evident software exist?
2. If so, under what model and with what assumptions?
3. Can tamper-evident software be developed to solve the hostile host problem?

To answer these questions we adopt a formal automaton model of computation. We model the hostile host by specifying a set of automatons that represent the possible states of a popular eight-bit micro-controller. We consider two communication models between the host and verifier: synchronous and asynchronous. The communication models formally specify the characteristics of the channels used to transmit messages between parties.

In our approach, the verifier cannot directly observer the contents of the host's memory or registers. Rather than assume direct low level access to a host, we attempt to infer the host's state from its responses to challenges. The primary difficulty that arises is that observing responses alone provides little information since malicious software on a host can simulate a computation and return a correct response.

To defend against simulation, we augment our computational model to included the time required to compute. We use time as a side-channel to infer additional information about the execution of a program.

Even with this additional information, inferring that a program's execution is not tampered with is difficult since a host can optimize a computation and efficiently simulate a correct response.

In this paper, we prove the existence of programs which through their own execution provide sufficient information in the form of responses and timing to detect tampering. We refer to such programs as *timed tamper-evident programs*. Modifications to the integrity of a timed tamper-evident program or its execution state are detectable by an external verifier without low level access to the host. The timed tamper-evident programs developed in this document resist efficient simulation by already being an optimal implementation of the response generation code. Hence, attempts to efficiently simulate their execution result in an increased runtime. The only way for a host to return a correct response in the correct amount of time is to allow the timed tamper-evident program to execute untampered.

Contributions.

- We formally define timed tamper-evident software.
- We prove the existence of timed tamper-evident software for an eight-bit micro-controller modeled using automatons under a synchronous communication model.
- We construct a timed tamper-evident program which provably guarantees data integrity for arbitrary memory locations.
- We construct a timed tamper-evident program which provable guarantees control integrity.
- We develop a general solution to the software-only hostile host problem by combining data and control tamper-evident programs.
- We discuss the impossibility of timed tamper-evident software in an asynchronous communication model.

Organization. This paper is organized as follows. Section 2 formally defines the problem. Section 3 contains our end-system computational model. Section 4 and contain our results. Section 5 discusses communication models and the existence of tamper-evident software in asynchronous systems. Section 6 discusses limitations and extensions of our approach. We describe related work in Section 7. Section 8 contains our conclusions and future work.

2 Formal Problem Definition

We model the state of a computer and its possible behaviors as an automaton. The automaton model is similar to a Moore state machine but augmented with a notation of time.

Definition 1 (Automaton)**.** *An* automaton *consists of a set of states S, a start state s_0, a set of inputs Σ, a set of outputs Γ, a transition function* $\mathsf{next} : S \times \Sigma \to S$ *(we assume that the machine is deterministic), an output function* $\mathsf{out} : S \to \Gamma$*, and a transition time function* $\mathsf{time} : S \times \Sigma \to \mathbb{N}$*, where* $\mathbb{N}$ *is the set of natural numbers.*

The transition time function assigns to a transition the amount of time (in cycles) it takes to complete. The notion of time complicates how the automaton interacts with its environment, that is, how it consumes its inputs. Unlike normal automaton models, the amount of time that an input is present can alter the behavior of the automaton. Thus, we model the user inputs not as simply a sequence of inputs, but rather as a function from a time in $\mathbb{N}$ to a input in Σ. Likewise, we treat the output as a function from $\mathbb{N}$ to Γ. (Although this model treats all automatons as though they never terminate, a distinguish output could be added to represent observable termination.)

We give an operational interpretation of our automatons. Let $M = \langle S, s_0, \Sigma, \Gamma, \mathsf{next}, \mathsf{out}, \mathsf{time}\rangle$ be an automaton, and $f_{\mathsf{in}} : \mathbb{N} \to \Sigma$ be an input function. Let s_j be $\mathsf{next}(s_{j-1}, f_{\mathsf{in}}(t_{j-1}))$ and t_j be $\mathsf{time}(s_{j-1}, f_{\mathsf{i}}(t_{j-1}))$ for all $j \geq 1$, and let $t_0 = 0$. We call s_j the current state at any time $t_j \leq \tau < t_{j+1}$. M subjected to f_{i} will produce the output function $f_{\mathsf{out}} : \mathbb{N} \to \Gamma$ such that for all $0 \geq j$, and all $t_j \leq \tau < t_{j+1}$, $f_{\mathsf{out}}(\tau) = \mathsf{out}(s_j)$.

We denote such an output function as $M(f_{\mathsf{in}})$. Given a output function f_{out} and a time $t \in \mathbb{N}$, we denote by $\lfloor o \rfloor_t$ the finite prefix of f_{out} up to time t. More formally, $\lfloor f_{\mathsf{out}} \rfloor_t$ is the t-element sequence of Γ^* such that for all $0 \leq \tau < t$, $\lfloor f_{\mathsf{out}} \rfloor_t(\tau) = f_{\mathsf{out}}(\tau)$ where $\lfloor f_{\mathsf{out}} \rfloor_t(\tau)$ denotes the τth element of $\lfloor f_{\mathsf{out}} \rfloor_t(\tau)$.

Given an input function f_{in} and two automatons M_1 and M_2 (such that the codomain of f_{in} is a subset of each of their set of inputs), let $M_1 \sim^t_{f_{\mathsf{in}}} M_2$ iff $\lfloor M_1(f_{\mathsf{in}}) \rfloor_t = \lfloor M_2(f_{\mathsf{in}}) \rfloor_t$.

Let $\mathsf{machine}(S, \Sigma, \Gamma, \mathsf{next}, \mathsf{out}, \mathsf{time})$ denote the set of automatons

$$\{ \langle S, s, \Sigma, \Gamma, \mathsf{next}, \mathsf{out}, \mathsf{time} \rangle \mid s \in S \}$$

Intuitively, $\mathsf{machine}(\Sigma, \Gamma, \mathsf{next}, \mathsf{out}, \mathsf{time})$ defines the hardware of a computer since it specifies the possible states and how the computer may operate, but says nothing about the current state of the machine (the software and data). We overload notation to allow $\mathsf{machine}(\Sigma, \Gamma, \mathsf{next}, \mathsf{out}, \mathsf{time})(s)$ to be $\langle S, s, \Sigma, \Gamma, \mathsf{next}, \mathsf{out}, \mathsf{time} \rangle$.

Definition 2 (Tamper-Evident State). *A state $s \in S$ is* tamper-evident *under an input function $f_{\mathsf{in}} : \mathbb{N} \to Sigma$ and time $t \in \mathbb{N}$ for a set of automatons $\mathcal{M} = \mathsf{machine}(S, \Sigma, \Gamma, \mathsf{next}, \mathsf{out}, \mathsf{time})$ iff there exists no $M \in \mathcal{M}$ other than $\mathcal{M}(s)$ such that $\mathcal{M}(s) \sim^t_{f_{\mathsf{in}}} M$.*

If $s \in S$ is a tamper-evident state, then given a automaton M drawn from a known set of automatons $\mathcal{M} = \mathsf{machine}(S, \Sigma, \Gamma, \mathsf{next}, \mathsf{out}, \mathsf{time})$, one may determine if $\mathcal{M}(s) = M$ even if one cannot directly inspect the start state of M. All one needs to do is supply M with the input function f_{in} and wait until the time t to see if $\lfloor M(f_{\mathsf{in}}) \rfloor_t$ is the correct output sequence that no other automaton of $\mathcal{M}$ can produce.

To define a *tamper-evident program*, we must refine the model of an automaton to the point where the meaning of a program becomes apparent. Let use restrict our attention to those automatons whose state space S is equal to $V^{\mathsf{m}} \times S'$ where $\mathsf{m} \in \mathbb{N}$. V^{m} represents the memory of the automaton, which has m locations. V is the set of values that a memory location may take on (e.g., 0 to $2^8 - 1$ in a 8-bit machine). S' represents the rest of the state of the automaton including things like the value of the program counter and any registers. We call these automatons *standard*.

We let $\mathsf{machine}'(V, \mathsf{m}, S', \Sigma, \Gamma, \mathsf{next}, \mathsf{time})$ represent those standard automatons that are in $\mathsf{machine}(V^{\mathsf{m}} \times S', \Sigma, \Gamma, \mathsf{next}, \mathsf{out}, \mathsf{time})$.

We say that $\vec{p} \in V^{\mathsf{p}}$ is a *program* of length p for $\mathsf{machine}'(V, \mathsf{m}, S', \Sigma, \Gamma, \mathsf{next}, \mathsf{time})$ if $\mathsf{p} \leq \mathsf{m}$. A given program $\vec{p}$ of length p for $\mathcal{M} = \mathsf{machine}'(V, \mathsf{m}, S', \Sigma, \Gamma, \mathsf{next}, \mathsf{time})$ is *loaded* at location ℓ on $M \in \mathcal{M}$ if the start state of M, $\langle \vec{m}, s' \rangle$, is such that $\vec{m}$ from ℓ to $\ell + \mathsf{p}$ is equal to $\vec{p}$ where $\ell + \mathsf{p} \leq \mathsf{m}$.

Definition 3 (Tamper-Evident Program). *Let $\vec{p}$ be a program of length p for the set of automatons $\mathcal{M} = \mathsf{machine}'(V, \mathsf{m}, S', \Sigma, \Gamma, \mathsf{next}, \mathsf{time})$. $\vec{p}$ is tamper-evident under the input function $f_{\mathsf{in}} : \mathbb{N} \to \Sigma$ and time $t \in \mathbb{N}$ at location ℓ for $\mathcal{M}$ iff for every M_p and $M_{\neg p}$ in $\mathcal{M}$, if M_p has p loaded at ℓ and $M_{\neg p}$ does not, then $M_p \not\sim^t_{f_{\mathsf{in}}} M_{\neg p}$.*

Intuitively such programs are tamper-evident since one may tell if the program is loaded at location ℓ or not. This definition may easily be extended to to deal with program that are not in contiguous memory locations. Indeed, the definition can also be extended to deal with components of state other than memory locations, such as the value of the program counter. The essence of this definition is simply that some subset of the state must be protected.

Finding a truly tamper-evident program may be difficult or even impossible for a given set of automatons. Furthermore, even if one can be found, it may have undesirable features like an extraordinary length or the time t may be vary long. Thus, we are interested in programs that are probabilistically tamper-evident.

Definition 4 (Probabilistically Tamper-Evident Program). *A program $\vec{p}$ for a set of automatons $\mathcal{M} = \mathsf{machine}'(V, \mathsf{r}, \mathsf{m}, \mathsf{next}, \mathsf{time})$ is probabilistically tamper-evident under the finite set of input functions $F_{\mathsf{in}} \subset \mathbb{N} \to V$ and time $t \in \mathbb{N}$ at location ℓ for $\mathcal{M}$ iff for every M_p that has $\vec{p}$ loaded at ℓ, there exists an $f_{\mathsf{in}} \in F_{\mathsf{in}}$ such that for every $M_{\neg p}$ that does not have $\vec{p}$ loaded at ℓ, then $M_p \not\sim^t_{f_{\mathsf{in}}} M_{\neg p}$.*

We call such a program probabilistically tamper-evident since given a M_p, for any $M_{\neg P}$ that an adversary can select, with some probability (bounded to be at least $1/|F_{\mathsf{in}}|$), the user will select a input function $f_{\mathsf{in}} \in F_{\mathsf{in}}$ such that $M_p \not\sim^t_{f_{\mathsf{in}}} M_{\neg p}$. This allows us to determine that $\vec{p}$ is loaded at ℓ with some probability.

The focus of this paper is finding probabilistically tamper-evident programs. For the rest of this paper, we use "probabilistically tamper-evident program" and "tamper-evident program" interchangeably.

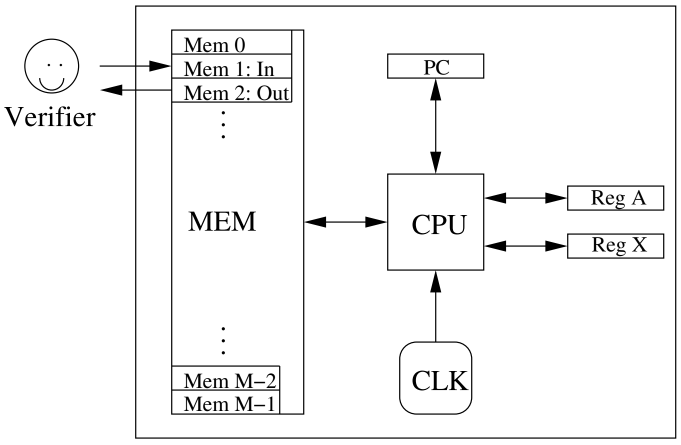

Figure 1: Architecture of the Motorola 68H(7)05H12

3 System Formalization

We focus our search for tamper-evident programs on one particular machine, the Motorola 68H(7)05H12. The architecture of this machine is summarized in Figure 1. This eight-bit micro-controller is capable of executing 62 instructions each of which come in up to eight different flavors corresponding to different addressing modes. Instructions take from two cycles to as many as eleven to execute. The instructions may load values from memory and make use of two 8-bit registers named A and X. In the formal model, the contents of this memory corresponds to the value $\vec{m}$ and contents of the registers to $\vec{r}$, which is of length two.

The chip communicates with the outside world through interrupts, resets, and I/O ports. All of these may be modeled as various forms of input. Since the tamper-evident programs we propose do not make use of interrupts and resets we do not formally model them. We model the I/O ports as two distinguished memory locations: one for input at memory location 1 and one for output at memory location 2. Thus, the value of the memory cell 2 determines the value of the out function at any given time.

The state is further defined by the value of a 14-bit program counter (PC), a 5-bit stack pointer, and five status flags. Considering all these elements results in the set of states being

$$\mathbb{Z}_{2^8} \times \mathbb{Z}_{2^8} \times \mathbb{Z}_{2^{14}} \times \mathbb{Z}_{2^5} \times \mathbb{Z}_2 \times \mathbb{Z}_2 \times \mathbb{Z}_2 \times \mathbb{Z}_2 \times \mathbb{Z}_2 \times \mathbb{Z}_{2^8}^{\mathsf{m}}$$

where m is the size of the memory and $\mathbb{Z}_n$ is $\{0, 1, \ldots, n-1\}$. The first component represents the register A; the second, the register X; the third, the PC; the forth, the stack pointer; the fifth to ninth represent the five flags; and the last component corresponds to the memory.

The transition function next is determined by the semantics of the instruction set. The details of these instructions may be found in the chip's specification [6]. We present three instructions in detail to show that we can in fact treat them formally. The other instructions are explained as they are encountered.

`NOP` First, consider the instruction `NOP`, the noop. In principle, it does nothing, however, a state change does take place when it is encountered. The PC is incremented once to move beyond it. Also, the current input value is moved into memory location 2.

These effects are captured by the transition function next as

$$\mathsf{next}(\langle a, x, p, s, f_1, f_2, f_3, f_4, f_5, \vec{m}\rangle, i) = \langle a, x, p+1, s, f_1, f_2, f_3, f_4, f_5, \vec{m}[2 \mapsto i]\rangle$$

if $\vec{m}(p) = 9\mathrm{D}$ where 9D is the opcode for `NOP` (in hex), $\vec{m}(y)$ indicates the yth element of $\vec{m}$, and $\vec{m}[2 \mapsto i]$ is like $\vec{m}$ except that $\vec{m}[2 \mapsto i](2)$ is i regardless of the value of $\vec{m}(2)$.

Similarly, the specification of `NOP` influences the behavior of the timing function. Since `NOP` takes two cycles to execute, time is such that

$$\mathsf{time}(\langle a, x, p, s, f_1, f_2, f_3, f_4, f_5, \vec{m}\rangle, i) = 2$$

if $\vec{m}(p) = 9\text{D}$.

`LDA/ix` The instruction `LDA/ix` is for loading values from memory into the register A. The instruction treats the contents of the register X as a memory address. `LDA/ix` loads into A the contents of the memory cell whose address is stored in the register X. The PC is also incremented and the input value is placed in memory location 2 as with the `NOP` instruction. More formally,

$$\mathsf{next}(\langle a, x, p, s, f_1, f_2, f_3, f_4, f_5, \vec{m}\rangle, i) = \langle \vec{m}(x), x, p+1, s, f_1, f_2, f_3, f_4, f_5, \vec{m}[2 \mapsto i]\rangle$$

if $\vec{m}(p) = \text{F6}$ where F6 is the opcode for `LDA/ix`.

Since `LDA/ix` takes three cycles to complete

$$\mathsf{time}(\langle a, x, p, s, f_1, f_2, f_3, f_4, f_5, \vec{m}\rangle, i) = 3$$

if $\vec{m}(p) = \text{F6}$.

There are five other instruction like `LDA/ix` that load a value into the register A. They differ in if they have an argument and if so how they treat the argument and argument's length. We refer to all these instructions collectively as LDA.

`LDA/dir` The instruction `LDA/dir` is one such LDA instruction. Unlike `NOP` and `LDA/ix`, `LDA/dir` takes a one byte argument. The argument is the next byte in memory following the instruction's opcode. This argument is treated as a memory address. The contents of the memory cell with this memory address is loaded into the register A by the end of the `LDA/dir` instruction. Since `LDA/dir` takes a one byte argument, the PC is incremented twice: once to move beyond the opcode and once to move beyond the argument. More formally,

$$\mathsf{next}(\langle a, x, p, s, f_1, f_2, f_3, f_4, f_5, \vec{m}\rangle, i) = \langle \vec{m}(\vec{m}(p+1)), x, p+2, s, f_1, f_2, f_3, f_4, f_5, \vec{m}[2 \mapsto i]\rangle$$

and

$$\mathsf{time}(\langle a, x, p, s, f_1, f_2, f_3, f_4, f_5, \vec{m}\rangle, i) = 3$$

if $\vec{m}(p) = \text{B6}$ where B6 is the opcode for `LDA/dir` since `LDA/dir` also takes three cycles to execute.

4 Tamper-Evident Programs

We study two classes of tamper-evident programs: tamper-evident programs which guarantee data integrity and tamper-evident programs which guarantee control integrity.

4.1 Classes of Tamper-Evident Programs

Definition 5. *A **data tamper-evident** program allows a verifier to verify the integrity of data at a specified memory location.*

Data tamper-evident programs allow a verifier to check that the correct data is stored at an arbitrary memory location, allowing the verification the contents of memory. Data tamper-evident programs enable software authentication.

Definition 6. *A **control tamper-evident** program allows a verifier to verify the integrity of the value of the program counter.*

Control tamper-evident programs allow a verifier to verify if the program counter is pointing at an expected address. When used in conjunction with data tamper-evident programs, control tamper-evident programs enable the verification of program and execution state integrity.

4.2 Data Tamper-Evident Programs

Consider the following program Q:

```
start: LDX/dir 1 ; 3 cycles, 2 bytes, load the contents of input to X
       STX/dir 2 ; 4 cycles, 2 bytes, store the value of X into output
print: LDA/ix    ; 3 cycles, 1 byte,  load into A the contents of memory at X's value
       STA/dir 2 ; 4 cycles, 2 bytes, store the value of A into output
       INCX      ; 3 cycles, 1 byte,  increment the value of X
       BNE print ; 3 cycles, 2 bytes, goto print if X is not 0
```

where `start` is initial value of the PC after a reset and `print` is the address 4 bytes beyond that. If Q starts at time 0, then its behavior is as follows: take an input i at time 0, at time 7 echo i to output, at time 14 print the contents of the memory location with address i, at time 27 (13 cycles later) print the contents of the memory location with address $i+1$, at time 40 (another 13 cycles later) print the contents of the memory location with address $i+2$, and so forth until value of the next address to be printed rolls over to 0. After that, the program's behavior is unspecified.

The program Q was shown by Gratzer and Naccache to be a data tamper-evident program when using resets to set the PC to `start` at a known time [2]. This, however, may e unrealistic since most chips have a measure of variance in how long it takes to identify and effect a reset (See Section 6.2 for details.). Thus, the time between triggering the reset and the PC actually reseting to `start` is not exactly known.

One way to minimize the impact of nondeterministic reset timing is to place Q in an interrupt handler instead of at the PC's initial value. This way, restarting the program by setting the PC to the program's first instruction only requires an interrupt instead of a more complex reset. This is a slight improvement since interrupts often have more deterministic timing properties.

However, we are interested in eliminating the need for either a reset or an interrupt since both suffer from nondeterministic timing on some chips. To do this we enumerate and eliminate each reason that a reset or interrupt is needed:

1. The first reason is that Gratzer and Naccache's proof of tamper-evidence requires that multiple executions of Q to eliminate the possibility that malicious code might simulate $Q's$ execution. If we are willing to accept a weaker probabilistic guarantee, then we may remove this requirement. That is, instead of running Q multiple times to guarantee that it is running, we can instead run it only once. Since an adversary cannot simulate all the possible behaviors of the program in the required amount of time, there is some probability that the input we provide is one of the ones the adversary cannot simulate. (See Definition 4 for a formalization.) If the probability of the adversary evading detection is negligible, this may be an acceptable solution. We have not, however, attempted to calculate this probability.

2. The second reason is that one needs a way to verifiable invoke Q. Before Q is invoked, the host will be executing some other program. In the case of sensor nodes, this might be a program that periodically measures the temperature of the surrounding environment. To execute Q, we must be able to stop the currently executing program and context switch to Q in way that allows the verifier to determine when Q started executing. Furthermore, invocation must work even if an adversary has tampered with the currently executing program.

   To enable timely invocation without relying on nondeterministic resets or interrupts, we augment the currently executing program in such a way that the verifier need not perform an interrupt or reset to invoke Q. We do not present a formal algorithm for rewriting all programs, but rather show an example of what such a rewritten program might look like. We assume that the normal program consists an outer loop that normally runs forever. We replace the unconditional jump at the end of such a loop with a branch statement that breaks out of the loop when given a special input value. The value signifies that Q should be executed. To allow for the timely invocation of Q, the code preceding the branch in the currently executing program alters the output value so that the verifier can determine where in the loop the program is. Such a program might look like:

```
begin: LDA/dir 3   ; 3 cycles, 2 bytes, load the value memory location 3 into A
       ...         ; the normal program
       STA/dir 3   ; store the value of A into memory location 3
       LDA/imm 0   ; 2 cycles, 2 bytes, put 0 into X
       STA/dir 2   ; 4 cycles, 2 bytes, store the value of A into output
       LDA/imm 1   ; 2 cycles, 2 bytes, put 1 into X
       STA/dir 2   ; 4 cycles, 2 bytes, store the value of A into output
       LDA/dir 1   ; 3 cycles, 2 bytes, load the input into A
       BNE begin   ; 3 cycles, 2 bytes, goto print if A is not 0
start: LDX/dir 1   ; 3 cycles, 2 bytes, load the contents of input to X
       ...         ; the rest of the program Q
```

For this code to work, we assume that memory location 3 is not used by the currently executing program allowing us to save the value of A in its location. We assume that the currently executing program does not print output (or at least not the values 0 or 1). The first step of the modified program is to restore the value of A from memory location 3, after which the body of the program is executed. Next, A is saved to memory location 3 and the value of the output register quickly changed from 0 to 1 to signify the current point in the execution of the loop. After noticing the output go to 0, the verifier loads the value 0 into the input location to invoke Q which is resident just below the branch statement.

Although we have not formally proved that the above scheme works, we will informally argue that is works by explaining how it resists three types of attacks:

- The adversary could alter the code so that it shows the verifier the transition of the output value from 0 to 1 at the wrong time. For this to help the adversary, it would have to result in his program getting a head start on the computation. However, there is no way for this to happen since after the verifier sets the input to 0 indicting that temper detection program should be ran, he withholds the next input for fixed period of time.
- The adversary could alter the code so that the Q does not start running and rather some other program runs. Such an attack, however, can be detected (with some probability) since as explained above, only Q has the required output and timing behavior.
- The adversary could alter the code so that while it will run Q, it will run a different copy of Q (or some other semantically equivalent program) that is somewhere else in memory. While this would result in different code being executed, this is a benign attack since this different program must be semantically equivalent to Q for it to have the correct behavior. That is, this program only differ in its location in memory.

3. This last attack brings us to the third reason that resets and interrupts are useful: Q is merely data tamper-evident and not control tamper-evident. That is, while we can be ensured that a program semantically equivalent to Q is running and we may check the contents of any location in memory (including those where Q is stored), we cannot be sure where the executing copy of Q is actually stored. Thus, an exogenous means for determining the control state of the machine, such as resets or interrupts, is required. We eliminate this requirement in the next section by developing a control tamper-evident program and discussing how to use it in conjunction with Q to detect either form of attack.

4.3 Control Tamper-Evident Programs

Consider the fragment code we call the Spring Board:

```
50: LDX/imm 126  ; 2 cycles, 2 bytes, put 126 into X
52: STX/dir 2    ; 4 cycles, 2 bytes, store the contents of X into output
54: JMP/dir 25   ; 2 cycles, 2 bytes, jump to the location 25
56: LDX/imm 42   ; 2 cycles, 2 bytes, put 42 into X
```

Input at time 0		Output at time 8	
decimal	binary	decimal	binary
50	00110010	126	01111110
56	00111000	42	00101010
62	00111110	36	00100100
68	01000100	68	01000100

Table 1: Possible Behaviors of the Launcher combined with the Spring Board.

```
58: STX/dir 2    ; 4 cycles, 2 bytes, store the contents of X into output
60: JMP/dir 25   ; 2 cycles, 2 bytes, jump to the location 25
62: LDX/imm 36   ; 2 cycles, 2 bytes, put 36 into X
64: STX/dir 2    ; 4 cycles, 2 bytes, store the contents of X into output
66: JMP/dir 25   ; 2 cycles, 2 bytes, jump to the location 25
68: LDX/imm 68   ; 2 cycles, 2 bytes, put 68 into X
70: STX/dir 2    ; 4 cycles, 2 bytes, store the contents of X into output
72: JMP/dir 25   ; 2 cycles, 2 bytes, jump to the location 25
```

We call the addresses 50, 56, 62, and 68 the targets of the Spring Board.

Consider the following instruction that we call the Launcher:

```
JMP/dir 1        ; 2 cycles, 2 bytes, jumps to the address stored at input
```

Suppose, that the PC is at the start of the Launcher at time 0 and the Spring Board is at the proper location in memory. Under these conditions, if at time 0, 50 is in the input location, at exactly eight cycles later, 126 will appear in the output location. If instead 56 is in the input location, then 42 will appear at exactly eight cycles later. If 62 is provided as input, then 38 will be provided in eight cycles; and 68 will produce 68. Table 1 summarizes these behaviors. Note that this program will have these behaviors regardless of where in memory the Launcher is stored as long as the PC points to it.

The Launcher combined with the Spring Board provides a way to provably set the value of the PC. This is because, as we will soon prove, to observe the above behaviors, a jump to one the targets of the Spring Board must occur. Once execution is handed over to the Spring Board, the value of the PC will become know since every jump to a target of the Spring Board results in a jump to the location 25 (8 cycles after arriving at one of the target).

Thus, if one is able to verify that the Spring Board is in the correct location of memory and one feeds the computer a challenge and the computer responds to that challenge correctly, one can be sure with a fair probability of the PC's value. The following theorem formalizes our claims.

Theorem 1. *Suppose we know that Spring Board is stored in memory as shown above but the value of the PC is unknown. Suppose that the value of register X is not 1. If at time 0, the PC is pointing at code that is capable of each of the above behaviors, then at time 10, the PC must be 25.*

Proof. What we need to prove is that any program that can show all three behaviors must jump to one of the targets at exactly cycle 2. This is sufficient since under this condition the PC will be set to location 25 at time 10 regardless of which target was jumped to.

To prove that any program will have to jump to one the targets at cycle 2, we must show that there is no other way to produce the correct output exactly 8 cycles after getting the input for every possible input value given at cycle 0.

To do this, we constrain the possible set of programs to the point were only code like the above program are left. We start by ruling out a few classes of programs:

1. Clearly, since the output depends on the input, any program that does not include an instruction accessing the input value may be ruled out.

2. Obviously, we can rule out any program that stores or alters the value of the output location to be anything other than 0 at cycles 1 through 7 and anything other than the correct output value since the verifier would notice such a deviation.

3. Any program that preforms a shift on the output location since 13, 45, and 76 are not shifts of 0 may be ruled out.

4. Thus, one the following instructions must be used as the last instruction since all other instructions capable of effecting the output have been ruled out above: `STA/dir 2`, `STA/ix` with X=2, `STX/dir 2`, or `STX/ix` with X=2. We call these instructions the *printing instructions*. Since the output is seen at exactly 8 cycles and all the printing instructions take 4 to 6 cycles, the printing instruction must start at cycle 4, cycle 3, or cycle 2. Thus, any program that does not have such an instruction at one these cycles may be ruled out.

5. Since all the printing instructions simply moves the value from one of the registers into the output location, the value that will be printed must first be in one of these registers. Thus, any program that has not stored the correct output in one of the registers by cycle 4 may be ruled out. We may now limit our attention to the possible sequences of instructions to store this value in one of the two registers.

6. At this point, the only possible sequences left are either two 2-cycles instructions or one 4-cycle instruction. (There are no 1-cycle instructions.) Thus, BCLR, BRCLR, BRSET, JSR, MUL, RTI, RTS, and SWI cannot be used for this code since they each take more than 4 cycles.

7. Since none of the remaining instructions depend of the value of the stack pointer, RSP may be replaced by a NOP, so we need not consider it.

8. A branch statement may be ruled out since they all take at least three cycles leaving no time to actually load a value into a register.

9. All the of the rotations may be ruled out since they all take at least three cycles leaving no time to actually load a value into a register.

10. All the arithmetic operators (ADD, ADC, SUB, etc.) may be ruled out since none of them transform the input values to the correct output value. Since they all take at least three cycles to operate on input, they would have to be useful. To see this note that since 68 goes to 68, such an instruction must be the identity on 68. However, it would have also be capable of taking 50 to 126. However, no instruction is both the identity on 68 and not the identity on 50.

11. The three logical operators (AND, ORA, EOR) logical may be ruled out as above. First note that there is only time for the input value to be operated on with the current value of A, that is, a value that is constant with the value of the input. Now we treat each case: AND cannot send 50 = 00110010 to 01111110 = 126. ORA cannot send 62 = 00111110 to 00100100 = 36. For EOR to send 68 to 68, the constant that being EORed with must be 00000000. However, this fails to produce the correct value for the other inputs.

12. Any instructions that only set a flag (BIT, CPX, etc.) may be ruled out or replaced by NOP since no instruction that uses them are left.

13. STA or STX may be ruled out since they take at least four cycles and would leave no time to load a value into a register.

14. NOP may be ruled out since, executing it would leave only two cycles. In the two remaining cycles, a value that depends on the input value would have to be loaded into a register since the NOP does not depend on the value of the input register. However, there is no way to do this in only two cycles.

15. Any programs that does not use either `LDX/imm` or `LDA/imm` maybe ruled out since these are the only remaining instructions that can load the a value into one the registers.

16. This leaves two cycles for reading in the input. The only instruction that can do this is `JMP/dir 1`. Thus, it must be used. Since it does not set the value of either register, it must be followed by an instruction that does (either `LDX/imm` or `LDA/imm`) meaning that it must be the first instruction.

Thus, we have determined that the first instruction in any program producing the correct results must be `JMP/dir 1`. Since this does not give it time to alter the value of the input, it must directly jump to the location provided as input. Since the verifier provides the address of the targets, the program is forced to jump to one them. Thus, we have pinned down the value of the PC. $\square$

As discussed above, this result may be used to give a probabilistic guarantee. Namely, under the preconditions of the theorem, if a correct behavior is seen, we know with some probability that the PC is indeed 25 at time 10. Although we do not formally prove this, the probability appears to be 3/4th of the time. (That is, the adversary has a 1/4 chance to tamper with execution.) By increasing the number of targets in the Spring Board, this probability can be increased.

4.4 Using Data and Control Tamper-Evident Programs Together

Q and the Spring Board may be used together to create a program that is both data and control tamper-evident. The idea would be to first run Q to ensure that the Spring Board is in place. This would be followed by handing control over the to Launcher would have to have to hand control over to one the targets of the Spring Board allowing the PC value to become known as proved in Section 4.3. Note that the state of the computer after the execution of Q will satisfy the preconditions of the theorem where time 0 becomes the time at which Q passes control to the Launcher. (Q will leave the value of X as 0, not 1.)

Although an adversary could have try to use some program other than Q, this does not matter since any such program would have to semantically equivalent to Q and thus will be sufficient to determine that the Spring Board is in place. The adversary could have control move from this Q-like program to some program other than the Launcher. However, again this does not matter since as shown in Section 4.3, any such program must jump to one of the targets of the Spring Board to avoid detection.

5 Communication Models

We discuss synchronous and asynchronous communication models which specify the characteristics of the communication channel between the verifier and host. Synchronous systems roughly correspond to trusted channels while asynchronous systems correspond to untrusted channels. We discuss the possibility that timed tamper-evident functions do not exist in asynchronous systems.

5.1 Synchronous Systems

A communication system is called synchronous if there exists a known maximum delay on message transmission. We use the term synchronous to describe a system with a known *minimum* delay on message transmission. Such a system may correspond to a situation where an initial lower bound on message transmission can be established between the prover and the verifier or where the infrastructure between the prover and verifier can be trusted. Example synchronous systems include systems directly connected using a trusted serial port or enterprises with trusted switches.

Definition 7. *A synchronous system is a 3-tuple (N, E, D) where*

- *N is a set of nodes,*
- *E is a set of edges or allowing communication between nodes, and*
- *D is a labeling function which assigns a minimum delay to each edge, defined as $D : E \to R^+$.*

5.2 Asynchronous Systems

A asynchronous system is a system without a known maximum delay on message transmission. In our formulation, an asynchronous system is a system without a known minimum delay on message transmission. Such a system may correspond to one where the host cannot be trusted to respond to delay measurement messages in a timely fashion or the infrastructure between the host and verifier cannot be trusted to provide an accurate lower bound on message transmission. Both of these cases are realistic when the communication system in question is not under the control of the verifier.

Definition 8. *An asynchronous system is a 3-tuple* (N, E, D) *where*

- N *is a set of nodes,*
- E *is a set of edges or channels between nodes, and*
- D *is a labeling function which assigns a delay interval to each edge and is defined as* $D : E \rightarrow (\mathbb{R}^+, \mathbb{R}^+)$

Our intuition is that timed tamper-evident programs do not exist in an asynchronous system. This results from the fact that asynchronous systems do not have a lower bound on message delays, making it impossible to distinguish malicious run-time overhead on the prover from network delay. We describe this situation more formally below:

Intuition. Let $A = (N, E, D)$ be an asynchronous system with prover $p \in N$, verifier $v \in N$, communication channel $c \in E$, and channel delay $D(c)$.

Assume a timed tamper-evident program T executes on P with expected response r and execution time τ, to show there exists a program T' that simulates T.

In order to identify tampering, the verifier must establish a minimum value for the execution of the program and the channel latency. Since the execution time is assumed to be known ahead of time, the verifier need only establish the channel delay, $D(c)$.

To measure the channel delay, P must send a message to V and await a response. P may iterate this process until satisfied that the delay is the minimum possible value. However, a malicious measurement program on P can execute the following code:

```
if (receive()) {        // Listen for measurement attempt
 wait(k seconds);       // Delay for k seconds
 respond();             // Reply to measurement attempt
}
```

Since V is unable to distinguish between delay induced by P and delay inherent in the transmission of a message, V is unable to ascertain an accurate timing of $D(c)$. The wait command allows $T's$ execution to be simulated to within k seconds by a malicious T'.

Given an arbitrary timed tamper-evident program T, we can produce a program with undetectable tampering denoted T'. Such a program may look similar to the following:

```
tamper(T);                    // Tamper with T's code
response = simulate(T, k/2);  // Simulate T within k/2 seconds
respond(response);            // Return response
```

While not a formal proof, we believe that timed tamper-evident programs do not exist in a asynchronous systems. We intended to more thoroughly explore this possibility in future work.

5.3 Alternative Communication Models

The existence of synchronous channels with a known minimum delay on message transmission is a strong assumption. However, without such channels, we are unable to establish the exact time elapsed during the execution of a tamper-evident program. To use timed tamper-evident functions over a computer network we need to relax our synchronous channel assumption to allow for the possibility of an unknown minimum delay or a known minimum delay which is rarely if ever achieved. We call such a channel a noisy synchronous channel.

Under the noisy synchronous channel model, malware can undetectable tamper with the execution of the tamper-evident program similar to the asynchronous case. A promising approach to defend against malicious tampering in such a scenario is to allow for tampering as long as the resulting program accomplishes the same task as the original. Previously, our definition of tampering was that the exact sequence of instructions must execute in the expected state in order for the execution to be considered untampered. If we relax this requirements to allow for benign modifications we may be able to overcome some channel noise while still providing a guarantee that semantically equivalent code executed. This is the model under which current tamper-evident software primitives such as Pioneer and SCUBA [8, 7] operate.

6 Discussion

6.1 The Hostile Host Problem

Our data and control tamper-evident programs allow for the establishment of a trusted computing base on an untrusted host. From this trusted computing base, one can bootstrap to verifiable code execution of arbitrary programs and the verification of memory integrity.

Verifiable Code Execution. Verifiable code execution is a process whereby a verifier receives from a prover a guarantee of (1) the integrity of a target program and (2) that the target code is invoked for execution in a protected environment on the prover. Achieving verifiable code execution is a two step process: the prover must (1) verify the integrity of the target code and (2) execute the target code in the protected environment. Verifying the integrity of the target code guarantees that the target code was not modified before execution and the protected environment guarantees that no other code interferes with the execution of the target code. When used in conjunction, these two steps guarantee the control ad data integrity of the target executable[1].

Agent on Hostile Host. Assume an execution agent exists on the prover which carries out the above steps. The execution agent first hashes the target code then sends the hash of the target code to the verifier, after which the target code is invoked for execution in the protected execution environment. Upon receipt of the hash value, the verifier checks the value against a previously computed value to ensure that the code image executed by the execution agent is unmodified. If the hash value is correct and the code executes in a protected environment after being invoked for execution, then the verifier obtains the guarantee of verifiable code execution.

Protected Execution Agent. To operate correctly, the execution agent on the prover must be protected. To guarantee verifiable code execution of arbitrary target code, the verifier requires a guarantee of verifiable code execution for the software execution agent itself. To address this cyclic-dependence, the data integrity of the execution agent can be checked by a data tamper-evident program which measures its own code integrity and the execution state of the prover. Similarly, the control integrity of the execution agent can be verified by a control tamper-evident program. In this way, our tamper-evident programs guarantee that tampering by a hostile host will be detected if it attempts to: (1) modify the execution agent before execution, (2) modify the execution agent during execution, or (3) modify the execution state.

6.2 Nondeterministic Timing

Our finite state model assumes the Motorola 68H(7)05H12 hardware is deterministically timed, however hardware often has some nondeterministic timing. For example, the time required for a signal to transition from low to high and the corresponding time for the processor to identify the transitions depends on the stage of the processor fetch and execute cycle that the processor is in when the signal transitions. Since this architectural state is not observable and is difficult to infer, we may not be able to achieve perfectly accurate reference times for instruction execution. Fortunately, the level of timing nondeterminism is often sub-cycle making it impossible for an adversary to execute additional instructions in the time required.

[1]Assuming there are no program vulnerabilities which can be exploited during execution and hence violate the guarantees of the protected execution environment.

6.3 Other Architectures

Our proof is based on the Motorola 68H(7)05H12 however, it may be possible to extend our proof to other architectures. For simple architectures like the Intel 8051, the complexity of extending our result primarily depends on the semantics of the instruction set and the timing of key instructions (e.g., load and store). With respect to timing, the instruction set timing of the Intel 8051 is more uniform that the Motorola 68H(7)05H12, meaning that the amount of side-channel information available is limited.

Suitability Metrics. A promising approach to measure the suitability of an architecture for timed tamper-evident functions is to compute the entropy of the instruction set times. For example, the Intel 8051 has 111 total instructions of which 64 execute in one cycle, 45 execute in two cycles, and 2 execute in four cycles. This gives the Intel 8051 a cycle entropy of $H = 1.09$, meaning that for each cycle observed 1.09 bits of information is transmitted through the timing side-channel. By measuring the cycle entropy for various architectures, we may be able to quickly compare architectures to determine the existence of timed tamper-evident functions on the architecture in question.

Entropy is not a perfect measure of the feasibility of implementing timed tamper-evident code on an architecture. It is trivial to show that by permuting the timing required by two instructions on the same architectures, hence holding the entropy constant, one may be well-suited to tamper-evident code while the other is not. Proving that tamper-evident program exist for a given architecture still probably requires an exhaustive search through all possible simulation options.

Modern Architectures. The complexity of modern architectures like the x86 makes manually extending our results both time consuming and tedious. Two problems arise when extending our results to modern architectures: (1) accurately modeling the complex functionality of the architecture is difficult and (2) brute-force search of all possible programs may be intractable.

To address the lather, we may be able to use automated search techniques such as refutation-based theorem provers to address the large number of cases that must be considered for the adversary's attempt to simulate our program. Other possibilities include using superoptimizers like Denali [4] which prove code optimality for small programs. However, accurately modeling modern architecture in sufficient detail to be able to reason about all possible attacks is likely a difficult task.

7 Related Work

The work closest to our own is that of Gratzer and Naccache [2]. Gratzer and Naccache prove that timed tamper-evident programs exist in a perfect timing model. However, Gratzer and Naccache required direct low level hardware support to ensure control integrity. The tamper-evident programs developed in this work eliminate the need for hardware support through a software technique.

Previous work has proposed that a verifier guarantee the memory integrity of a remote machine by interacting with a software-only tamper-evident program on the remote machine (see e.g., [5, 9, 8, 7]). The proposed tamper-evident program is stated to be time consuming to simulate. To ensure that the computer has not been compromised the verifier determines if the tamper-evident program is running or if malware is simulating it by timing the computer's response time to challenges. If the program's responses also relate information about the state of the computer (status registers, PC, etc..), then the verifier can be assured that the remote state is unadulterated.

While previous work in this area has presented code for such tamper-evident programs, they fail to offer a proof that any of these attempts are in fact time-intensive to simulate. As a result, one cannot be sure that the system really is in a known state. Rather one only knows that the author of the conjectured tampered-evident program was unable to simulate the program efficiently. As shown by vulnerabilities in previously proposed systems, this clearly does not prove that an attacker cannot find a way [10]. Providing a proof provides the user with guarantees rather than only hope.

8 Conclusion and Future Work

We have shown that timed tamper-evident software exists in a finite state model of computation under the assumption of a synchronous communication channel. Our solution required the strong assumptions of cycle-accurate timing and a known minimum delay on the channel between the verifier and host.

By constructing timed tamper-evident programs which guarantee data integrity and control integrity, we were able to bootstrap the untampered execution of arbitrary programs on a hostile host. Our technique is not without limitations, we require full knowledge of the hardware of the host and require that the hardware not be modified. Analyzing stronger adversarial models is a subject of future work.

Other subjects of future work include proving the existence of tamper-evident functions on modern architectures such as the ARM or x86. Proofs for such architectures probably require the assistance of computerized search techniques. By relaxing the definition of tamper-evident to include all semantically equivalent programs and accepting a probabilistic guarantee, we maybe able to eliminate the need for cycle-accurate timing and relax the delay requirements on the channel. One possible approach is to adopt a model similar to that of an zero knowledge proof system whereby a verifier obtains a probabilistic guarantee of a host's knowledge of some fact that is nearly impossible to consistently falsify, yet impossible to prove absolutely.

References

[1] Secure virtual machine architecture reference manual. AMD Corp., May 2005.

[2] Vanessa Gratzer and David Naccache. Alien vs. quine, the vanishing circuit and other tales from the industry's crypt. In *Proceedings of Eurocrypt 2006*, May 2006.

[3] Intel Corp. *LaGrande Technology Architectural Overview*, September 2003.

[4] R. Joshi, G. Nelson, and K. Randall. Denali: a goal-directed superoptimizer. In *Proceedings of ACM Conference on Programming Language Design and Implementation (PLDI)*, 2002.

[5] Rick Kennell and Leah H. Jamieson. Establishing the genuinity of remote computer systems. In *Proceedings of the 2003 USENIX Security Symposium*, August 2003.

[6] Motorola. *68HC(7)05H12: General Release Specification.* Motorola, November 1998. Revision 1.0.

[7] Arvind Seshadri, Mark Luk, Adrian Perrig, and Pradeep Khosla. SCUBA: Secure code update by attestation in sensor networks. In *Proceedings of the ACM Workshop on Wireless Security (WiSe)*, September 2006.

[8] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy platforms. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, October 2005.

[9] Arvind Seshadri, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. SWATT: Software-based attestation for embedded devices. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2004.

[10] Umesh Shankar, Monica Chew, and J.D. Tygar. Side effects are not sufficient to authenticate software. In *Proceedings of the 2004 USENIX Security Symposium*, August 2004.

[11] Trusted Computing Group (TCG). `https://www.trustedcomputinggroup.org/`, 2003.