Alien *vs.* Quine, the Vanishing Circuit and Other Tales from the Industry's Crypt

Vanessa Gratzer[1] and David Naccache[1,2]

[1] Université Paris II Panthéon-Assas, Hall Goullencourt, casier 55,
12 place du Panthéon, F-75231, Paris, CEDEX 05, France
`vanessa@gratzer.fr`

[2] École Normale Supérieure, Équipe de Cryptographie,
45 rue d'Ulm, F-75230, Paris, CEDEX 05, France
`david.naccache@ens.fr`

Abstract. This talk illustrates the everyday challenges met by embedded security practitioners by five real examples. All the examples were actually encountered while designing, developing or evaluating commercial products.

This note, which is not a refereed research paper, presents the details of one of these five examples. It is intended to help the audience follow that part of our presentation.

1 Foreword

When I was asked to give this talk, I was delighted, but a bit concerned.

What in my brief decade in the card industry would be of interest to a group of practitioners far more experienced in security than myself?

What will my story be?

As I started to question ex-colleagues, competitors and suppliers, I quickly realized that the problem would be in deciding what to leave out rather than what to include. I was finally able to narrow my list to five examples.

The first ones will deal with an electronic circuit that mysteriously vanished into thin air, DES *and* RSA *key-management in early-generation cards, a cryptographic watchdog chasing own tail and the story of the industry's first on-board sensors.*

This note, which is not a refereed paper, presents the details of the fifth example – coauthored with one of my students. It is intended to help the audience follow that part of the talk – a talk that I dedicate to the memory of our friends and colleagues Prof. Dr. Thomas Beth (1949–2005) and Prof. Dr. Hans Dobbertin, (1952–2006).

David Naccache

2 Introduction

Aliens are a fictional bloodthirsty species from deep space that reproduce as parasites. Aliens lay eggs that release araneomorph creatures (facehuggers) when

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-3-540-34547-3_36

S. Vaudenay (Ed.): EUROCRYPT 2006, LNCS 4004, pp. 48–58, 2006.

a potential host comes near. The facehugger slides a tubular organ down the victim's throat, implanting a larva in the victim's stomach.

Within a matter of hours the larva evolves into a chestburster and emerges, violently killing the host; chestbursters develop quickly and the cycle restarts.

Just as Aliens, rootkits, worms, trojans and viruses penetrate healthy systems and, once in, alter the host's phenotype or destroy its contents. Put differently, malware covertly inhabits *seemingly normal* systems until something triggers their awakening.

As illustrated recently [4], detecting new malware species may be a nontrivial task. In theory, the easiest way to exterminate malware is a disk reformat followed by an OS reinstallation from a trusted distribution CD. This relies on the assumption that computers can be *forced* to boot from trusted media.

However, most modern PCs have a flash BIOS. This means that the code-component in charge of booting has been recorded on a rewritable memory chip that can be updated by specific programs called *flashers* or, sometimes, by malware such as the CIH (Tchernobyl) virus.

Hence, a natural question arises:

How can we ascertain that malware did not re-flash the BIOS *to derail disk reformatting attempts and simulate their successful completion?*

Flash smart cards[1] are equally problematic. Consider a SIM-card produced by Alice and sold empty to Bob. Bob keys the card. Alice reveals an OS code but flashes a malware simulating the legitimate OS. When some trigger-event occurs[2] the malware responds (to Alice) by revealing Bob's keys.

This note describes methods allowing Bob to check that SIMs bought from Alice contain no malware. Bob's only assumption is that his knowledge of the device's *hardware* specifications is correct.

In biology, the term *Alien* refers to organisms introduced into a foreign locale. Alien species usually wreak havoc on their new ecosystems – where they have no natural predators. In many cases, humans deliberately introduce matching predators to eradicate the alien species. This is the approach taken here.

Related topic. What we try to achieve differs fundamentally from program competitions for the control of a virtual computer, such as Core War. Here the verifier *cannot see* what happens inside a device and seeks to infer the machine's state given its behavior.

3 The Arena

We tested the approach on Motorola's 68HC05, a very common eight-bit micro-controller (more than five billion units sold). The chip's specifications were very slightly modified to better reflect the behavior of a miniature PC.

[1] *e.g.* SST Emosyn, Atmel AT90SC3232, Infineon SLE88CFX4000P, Electronic Marin's EMTCG, *etc.*

[2] *e.g.* a specific 128-bit challenge value sent during the GSM authentication protocol.

The 68HC05 has an accumulator `A`, an index register `X`, a program counter `PC` (pointing to the memory instruction being executed), a carry flag `C` and a zero flag `Z` indicating if the last operation resulted in a zero or not. We denote by $\zeta(x)$ a function returning one if $x = 0$ and zero otherwise (*e.g.* $\zeta(x) = \lfloor 2^{-x} \rfloor$).

The platform has $\ell \leq 2^{16} = 65536$ memory bytes denoted $\texttt{M}[0], \ldots, \texttt{M}[\ell - 1]$. Any address $a \geq \ell$ is interpreted as $a \bmod \ell$. We model the memory as a state machine insensitive to power-off. This means that upon shut-down, execution halts and the machine's RAM is backed-up in non-volatile memory. Reboot restores RAM, resets `A`, `X`, `C` and `Z` and launches execution at address `0x0002` (which *alias* is `start`).

The very first RAM state (digital *genotype*) is recorded by the manufacturer in the non-volatile memory. Then the device starts evolving and modifies its code and data as it interacts with the external world.

The machine has two I/O ports (bytes) denoted `In` and `Out`. Reading `In` allows a program to receive data from outside while assigning a value to `Out` displays this value outside the machine. `In` and `Out` are located at memory cells $\texttt{M}[0]$ and $\texttt{M}[1]$ respectively. `Out`'s value is restored upon reboot (`In` isn't). If the device attempts to write into `In`, execute `In` or execute `Out`, execution halts.

The (potentially infested) system pretends to implement an OS function named `Install(`p`)`. When given a string p, `Install(`p`)` installs p at `start`. We do not exclude the possibility that `Install` might be modified, mimicked or spied by malware. Given that the next reboot will grant p complete control over the chip, `Install` would typically require some cryptographic proof before installing p.

We reproduce here some of the 68HC05's instructions (for the entire set see [3]). β denotes the function allowing to encode short-range jumps [3].

EFFECT	`lda` i	`sta` i	`bne` k	`bra` k
new `A` ←	$\texttt{M}[i \bmod \ell]$			
new `X` ←				
new `Z` ←	$\zeta(\text{new } \texttt{A})$	$\zeta(\texttt{A})$		
EFFECT ON `M`		$\texttt{M}[i \bmod \ell] \leftarrow \texttt{A}$		
new `PC` ←	$\texttt{PC} + 2 \bmod \ell$	$\texttt{PC} + 2 \bmod \ell$	$\beta(\texttt{PC}, \texttt{Z}, k, \ell)$	$\beta(\texttt{PC}, 0, k, \ell)$
OPCODE	0xB6	0xB7	0x26	0x20
CYCLES	3	4	3	3

[3] The seventh bit of k indicates if $k \bmod 128$ should be regarded as positive or negative, *i.e.*

$$\beta(\texttt{PC}, z, k, \ell) = \left(\texttt{PC} + 2 + (1 - z) \times \left(k - 256 \times \left\lfloor \frac{k}{128} \right\rfloor\right)\right) \bmod \ell.$$

EFFECT	`inca`	`incx`	`lda ,X`	`ldx ,X`
new $\mathtt{A}$ ←	$\mathtt{A}+1 \bmod 256$		$\mathtt{M}[\mathtt{X}]$	
new $\mathtt{X}$ ←		$\mathtt{X}+1 \bmod 256$		$\mathtt{M}[\mathtt{X}]$
new $\mathtt{Z}$ ←	$\zeta(\text{new } \mathtt{A})$	$\zeta(\text{new } \mathtt{X})$	$\zeta(\text{new } \mathtt{A})$	$\zeta(\text{new } \mathtt{X})$
EFFECT ON $\mathtt{M}$				
new $\mathtt{PC}$ ←	$\mathtt{PC}+1 \bmod \ell$	$\mathtt{PC}+1 \bmod \ell$	$\mathtt{PC}+1 \bmod \ell$	$\mathtt{PC}+1 \bmod \ell$
OPCODE	`0x4C`	`0x5C`	`0xF6`	`0xFE`
CYCLES	3	3	3	3

EFFECT	`ldx` i	`sta` i,`X`	`lda` i,`X`	`tst` i
new $\mathtt{A}$ ←			$\mathtt{M}[i+\mathtt{X} \bmod \ell]$	
new $\mathtt{X}$ ←	$\mathtt{M}[i \bmod \ell]$			
new $\mathtt{Z}$ ←	$\zeta(\text{new } \mathtt{X})$	$\zeta(\mathtt{A})$	$\zeta(\text{new } \mathtt{A})$	$\zeta(\mathtt{M}[i \bmod \ell])$
EFFECT ON $\mathtt{M}$		$\mathtt{M}[i+\mathtt{X} \bmod \ell] \leftarrow \mathtt{A}$		
new $\mathtt{PC}$ ←	$\mathtt{PC}+2 \bmod \ell$	$\mathtt{PC}+2 \bmod \ell$	$\mathtt{PC}+2 \bmod \ell$	$\mathtt{PC}+2 \bmod \ell$
OPCODE	`0xBE`	`0xE7`	`0xE6`	`0x3D`
CYCLES	3	5	4	4

EFFECT	`ora` i	`inc` i	`stx` i
new $\mathtt{A}$ ←	$\mathtt{A} \vee \mathtt{M}[i \bmod \ell]$		
new $\mathtt{X}$ ←			$\zeta(\mathtt{X})$
new $\mathtt{Z}$ ←	$\zeta(\text{new } \mathtt{A})$	$\zeta(\text{new } \mathtt{M}[i \bmod \ell])$	
EFFECT ON $\mathtt{M}$		$\mathtt{M}[i \bmod \ell] \leftarrow \mathtt{M}[i \bmod \ell]+1 \bmod 256$	$\mathtt{M}[i \bmod \ell] \leftarrow \mathtt{X}$
new $\mathtt{PC}$ ←	$\mathtt{PC}+2 \bmod \ell$	$\mathtt{PC}+2 \bmod \ell$	$\mathtt{PC}+2 \bmod \ell$
OPCODE	`0xBA`	`0x3C`	`0xBF`
CYCLES	3	5	4

4 Quines as Malware Predators

A Quine (named after the logician Willard van Orman Quine) is a program that prints a copy of its own code [1, 2]. Writing Quines is a tricky programming exercise yielding Lisp, C or natural language examples such as:

```
((lambda (x) (list x (list (quote quote) x)))
 (quote (lambda (x) (list x (list (quote quote) x)))))
```

```
char *f="char*f=%c%s%c;main(){printf(f,34,f,34,10);}%c";
main() {printf(f,34,f,34,10);}
```

```
Copy the next sentence twice. Copy the next sentence twice.
```

We start by loading a Quine into the tested computer. The device might be under the malware's total spell. The malware might hence neutralize the Quine or even analyze it and mutate (adapt its own code in an attempt to fool the verifier). As download ends, we start a protocol, called *phenotyping*, with whatever survived inside the platform.

Phenotyping will allow us to *prove* (Section 5) or assess the conjecture (Section 4) that the Quine survived and is now in full control of the platform. If the Quine survived we use it to reinstall the OS and eliminate itself; otherwise we *know* that the platform is infected. As we make no assumptions on the malware's malefic abilities, there exist extreme situations where decontamination *by software* is impossible. A trivial case is a malware controlling the I/O port and not letting anything new in. Under such extreme circumstances the algorithms presented in this note will only *detect* the malware but will be of no avail to eliminate it.

The underlying idea is that, upon activation, the Quine will (allegedly!) start dumping-out its own code plus whatever else found on board. We then *prove* or conjecture that the *unique* program capable of such a behavior, under specific complexity constraints, is *only* the Quine itself.

In several aspects, the setting is analogous to the scenario of *Alien vs. Predator*, where a group of humans (OS and legitimate applications) finds itself in the middle of a brutal war between two alien species (malware, Quine) in a confined environment (68HC05).

5 Space-Constrained Quines

We start by analyzing the simple Quine given below (`Quine1.asm`). This 19-byte program inspects $\ell = 256$ bytes platforms. `Quine1` is divided into three functional blocks separated by artificial horizontal lines. First, a primitive command dispatcher reads a byte from `In` and determines if the verifier wants to read the device's contents (`In` $= 0$) or write a byte into the RAM (`In` $\neq 0$).

As the program enters `print` the index register is null. `print` is a simple loop causing 256 bytes to be sent out of the device. As the loop ends, the device re-jumps to `start` to interpret a new command.

The `store` block queries a byte from the verifier, stores it in `M`[`X`] and re-jumps to `start`.

```
start:  ldx   In      ; X←In                0xBE 0x00
        bne   store   ; if X≠0 goto store   0x26 0x09
print:  lda   M,X     ; A←M[X]              0xE6 0x00
        sta   Out     ; Out←A               0xB7 0x01
        incx          ; X++                 0x5C
        bne   print   ; if X≠0 goto print   0x26 0xF9
        bra   start   ; if X=0 goto start   0x20 0xF3
store:  lda   In      ; A←In                0xB6 0x00
        sta   M,X     ; M[X]←A              0xE7 0x00
        bra   start   ; goto start          0x20 0xED
```

The associated phenotyping ϕ_1 is the following:

1. `Install`(`Quine1.asm`) and reboot.
2. Feed `Quine1` with 235 random bytes to be `store`d at `M`[21], . . . , `M`[255].

3. Activate `print` (command zero) and compare the observed output to:

$s_1 =$
```
0x00 0x00 0xBE 0x00 0x26 0x09 0xE6 0x00 0xB7 0x01
0x5C 0x26 0xF9 0x20 0xF3 0xB6 0x00 0xE7 0x00 0x20
0xED M[21],...,M[255]
```

Is `Quine1.asm` the only nineteen-byte program capable of always printing s_1 when subject to ϕ_1?

We conjecture so although (unlike the variant presented in the next section) we are unable to provide a formal proof. To illustrate the difficulty, consider a *slight* variant:

```
start: ldx   In      ; X←In                     0xBE 0x00
       bne   store   ; if X≠0 goto store        0x26 0x0B
label: tst   label   ;                          0x3D 0x06
---------------------------------------------------------
print: lda   M,X     ; A←M[X]                   0xE6 0x00
        ⋮     ⋮
                     ; same code as in Quine1
```

For all practical purposes, this modification (`Quine2.asm`)[4] has nearly no effect on the program's behavior: instead of printing s_1, this code will print:

$s_2 =$
```
0x00 0x00 0xBE 0x00 0x26 0x0B 0x3D 0x06 0xE6 0x00
0xB7 0x01 0x5C 0x26 0xF9 0x20 0xF1 0xB6 0x00 0xE7
0x00 0x20 0xEB M[23],...,M[255]
```

Let `Quine3` be `Quine2` where `tst` is replaced by `inc`.

When executed, `inc` will increment the memory cell at address `label` which is *precisely* `inc`'s own opcode. But since `inc`'s opcode is `0x3C`, execution will transform `0x3C` into `0x3D` which is... the opcode of `tst`.

All in all, ϕ_2 does not allow to distinguish a `tst` from an `inc` present at `label`, as both `Quine2` be `Quine3` will output s_2.

The subtlety of this example shows that a microprocessor-Quine-phenotyping triple $\{\mu, Q, \phi\}$ rigorously defines a problem:

> *Given a state machine μ find a state M (malware) that simulates the behavior of a state Q (legitimate* OS*) when μ is subject to stimulus ϕ (phenotyping).*

Security practitioners can proceed by analogy to the assessment of cryptosystems which specifications are published and submitted to public scrutiny. If an M simulating Q with respect to ϕ is found, a fix can either replace Q or ϕ or both. Note the analogy: *Given a stream-cipher μ and a key Q (defining an observed cipher-steam ϕ), prove that the key Q has no equivalent-keys M.*

An alternative solution, described in the next section, consists in *proving* the Quine's behavior under the assumption that the verifier is allowed to count clock cycles (state transitions if μ is a Turing Machine).

[4] ϕ_1 should be slightly twitched as well (233 random values to write).

6 Time-Constrained Quines

Consider the following program loaded at address `start`:

```
start: ldx In    ; 3 cycles ; X←In               (instruction $I_1$)
       stx Out   ; 4 cycles ; Out←X              (instruction $I_2$)
        ⋮   ⋮    ;          ; other instructions
```

Latch a first value v_1 at `In` and reboot, as seven cycles elapse v_1 pops-up at `Out`. If we power-off the device before the eighth cycle and reboot, v_1 reappears on `Out`[5] immediately. Repeating the process with values v_2 and v_3, we witness two seven-cycle transitions $v_1 \rightsquigarrow v_2$ and $v_2 \rightsquigarrow v_3$.

It is impossible to modify two memory cells in seven cycles as all instructions capable of modifying a memory cell require at least four cycles. Hence we are assured that between successive reboots, the *only* memory changes are in `Out`. This means that no matter what the examined code is, this code has no time to mutate in seven cycles and necessarily remains invariant between reboots.

The instructions other than `sta` and `stx` capable of modifying `dir`ectly `Out` are: `ror`, `rol`, `neg`, `lsr`, `lsl`, `asl`, `asr`, `bset`, `bclr`, `clr`, `com`, `dec` and `inc`. Hence, it suffices to select $v_2 \neq \mathtt{dir}(v_1)$ and $v_3 \neq \mathtt{dir}(v_2)$, where `dir` stands for any of the previous instructions[6], to ascertain that `Out` is being modified by an `sta` or an `stx` (we also need $v_1 \neq v_2 \neq v_3$ to actually *see* the transition).

$v_1 =$ `0x04`, $v_2 =$ `0x07`, $v_3 =$ `0x10` satisfy these constraints.

As reading or computing with a memory cell takes at least three cycles there are only four cycles left to alter the contents of `Out`; consequently, the only `sta` and `stx` instructions capable of causing the transitions fast enough are:

$I_2 \in$

sta Out	stx Out	sta ,X	stx, X

To aim at `Out` (which address is `0x0001`), `sta ,X` and `stx ,X` would require an `X=0x01` but this is impossible (if the code takes the time to assign a value to `X` it wouldn't be able to compute the transition's value by time). Hence, we infer that the code's structure is:

```
start: ??? ???     ; 3 cycles ; an instruction causing ● ←In
       st● Out     ; 4 cycles ; an instruction causing Out← ●
        ⋮   ⋮      ;          ; other instructions
```

where ● stands for register `A` or register `X`. The only possible code fragments capable of doing so are:

I_1 I_2 $\in$

adc In sta Out	adc ,X sta Out	add In sta Out	add ,X sta Out	eor In sta Out	eor ,X sta Out
lda In sta Out	lda ,X sta Out	ora In sta Out	ora ,X sta Out	ldx In stx Out	ldx ,X stx Out

[5] `Out` being a memory cell, its value is backed-up upon power-off.

[6] for `ror` and `rol`, consider the two sub-cases `C` $= 0$ and `C` $= 1$.

There is no way to further refine the analysis without more experiments, but one can already guarantee that as the execution of any of these fragments ends, the machine's state is either $\mathcal{S}_{\mathtt{A}} = \{\mathtt{A} = v_3, \mathtt{X} = \mathtt{0x00}\}$ or $\mathcal{S}_{\mathtt{X}} = \{\mathtt{A}=\mathtt{0x00}, \mathtt{X} = v_3\}$.

Now assume that $\mathtt{Out} = v_3 = \mathtt{0x10}$. Consider the code:

```
start: ldx In      ; 3 cycles ; X←In
       stx Out     ; 4 cycles ; Out←X
       lda ,X      ; 3 cycles ; A←M[X]    (instruction I3)
       sta Out     ; 4 cycles ; Out←A     (instruction I4)
        ⋮   ⋮      ;          ; other instructions
```

- Latch $\mathtt{In} \leftarrow v_4 = \mathtt{0x02}$, reboot, wait fourteen cycles; witness the transition[7] $\mathtt{0x10} \rightsquigarrow \mathtt{0x02} \rightsquigarrow \mathtt{0xBE}$; power-off before the fifteenth cycle completes.
- Latch $\mathtt{In} \leftarrow v_6 = \mathtt{0x04}$, reboot, wait fourteen cycles; witness the transition[8] $\mathtt{0xBE} \rightsquigarrow \mathtt{0x06} \rightsquigarrow \mathtt{0xF6}$; power-off before the fifteenth cycle completes.

As $v_5 \neq \mathtt{dir}(v_4)$ and $v_7 \neq \mathtt{dir}(v_6)$ the second transition is, again, necessarily caused by some member of the sta or stx families and, more specifically[9] one of the following:

$\mathtt{I}_4 \in$

sta Out	stx Out	sta ,X

$\mathtt{I}_3$ cannot be an instruction that has no effect on X and A as this will either inhibit a transition or cause a transition to zero (remember: immediately before the execution of $\mathtt{I}_3$ the machine's state is either $\mathcal{S}_{\mathtt{A}}$ or $\mathcal{S}_{\mathtt{X}}$). This rules-out eighteen jump instructions as well as all cmp, bit, cpx, tsta and tstx variants. lda i and ldx i are impossible as both would have forced 0x02 and 0x04 to transit to the *same* constant value.

In addition, $v_5 \neq \mathtt{dir}(v_4)$ implies that $\mathtt{I}_3$ cannot be a dir-variant operating on A or X, which rules-out negx, nega, comx, coma, rorx, rora, rolx, rola, decx, deca, dec, incx, inca, clrx, clra, lsrx, lsra, lslx, lsla, aslx, asla, asrx and asra altogether.

As no carry was set, we sieve-out sbc and adc whose effects will be strictly identical to sub i and add i (dealt with below).

add i, sub i, eor i, and i and ora i are impossible as the system

$$\begin{cases} \mathtt{0x02} \star x = \mathtt{0xBE} \\ \mathtt{0x06} \star x = \mathtt{0xF6} \end{cases}$$

has no solutions when operator $\star$ is substituted by $+, -, \oplus, \wedge$ or $\vee$.

The only possible $\mathtt{I}_3$ candidates at this point are:

$\mathtt{I}_3 \in$

sub ,X	and ,X	eor ,X	ora ,X	add ,X	lda, X	ldx ,X

[7] $v_5 = \mathtt{0xBE}$ is the opcode of ldx, read from address 0x02.

[8] $v_7 = \mathtt{0xF6}$ is the opcode of lda ,X, read from address 0x06.

[9] taking timing constraints into account and ruling-out stx ,X who can only cause an $\mathtt{Out} = \mathtt{0x01}$, a value never witnessed.

But before the execution of $\texttt{I}_3$ the machine's state is:

$$\mathcal{S}_{\texttt{A}} = \{\texttt{A} = \texttt{0x06}, \texttt{X} = \texttt{0x00}\} \text{ or } \mathcal{S}_{\texttt{X}} = \{\texttt{A} = \texttt{0x00}, \texttt{X} = \texttt{0x06}\}$$

The `",X"` versions of `sub`, `and`, `eor`, `ora` and `add` are impossible because:

- if the device is in state $\mathcal{S}_{\texttt{A}}$ we note that

$$\texttt{0x06} \star \texttt{0x06} \neq \texttt{0xF6} \text{ for } \star \in \{-, \vee, \oplus, \wedge +\}$$

- and if the device is in state $\mathcal{S}_{\texttt{X}}$ we note that

$$\begin{aligned}
\texttt{A} - \text{opcode}(\texttt{sub}, \texttt{X}) &= \texttt{0x00} - \texttt{0xF0} = \texttt{0x10} \neq \texttt{0xF6}\\
\texttt{A} \wedge \text{opcode}(\texttt{and}, \texttt{X}) &= \texttt{0x00} \wedge \texttt{0xF4} = \texttt{0x00} \neq \texttt{0xF6}\\
\texttt{A} \oplus \text{opcode}(\texttt{eor}, \texttt{X}) &= \texttt{0x00} \oplus \texttt{0xF8} = \texttt{0xF8} \neq \texttt{0xF6}\\
\texttt{A} \vee \text{opcode}(\texttt{ora}, \texttt{X}) &= \texttt{0x00} \vee \texttt{0xFA} = \texttt{0xFA} \neq \texttt{0xF6}\\
\texttt{A} + \text{opcode}(\texttt{add}, \texttt{X}) &= \texttt{0x00} + \texttt{0xFB} = \texttt{0xFB} \neq \texttt{0xF6}
\end{aligned}$$

`ldx ,X` is impossible as it would have caused a transition to $\text{opcode}(\texttt{ldx}, \texttt{X}) = \texttt{0xFE} \neq \texttt{0xF6}$ (if $\mathcal{S}_{\texttt{X}}$) or to `0x06` (if $\mathcal{S}_{\texttt{A}}$).

$\texttt{I}_3$ is hence identified as being necessarily `lda ,X`.

It follows immediately that $\texttt{I}_4 = $ `sta Out` and that the ten register-`A`-type candidates for $\{\texttt{I}_1, \texttt{I}_2\}$ are inconsistent.

The phenotyped code is thus one of the following two:

```
ldx In              ldx ,X
       ↘          ↙
         stx Out
         lda ,X
         sta Out
```

Only the leftmost is capable of causing the observed transition $\texttt{0x02} \rightsquigarrow \texttt{0xBE}$.

All in all, we have built a *proof* that the device actually executed the fragment presented at the beginning of this section.

Extending the code further ahead to:

```
start: ldx  In        ; X←In                  0xBE 0x00
       stx  Out       ; Out←X                 0xBF 0x01
print: lda  ,X        ; A←M[X]                0xF6
       sta  Out       ; Out←A                 0xB7 0x01
       incx           ; X ← X + 1             0x5C
       bne  print     ; if X ≠ 0 goto print   0x26 0xFA
```

and subjecting the chip to three additional experiments, we observe:

$$\begin{aligned}
\texttt{In} \leftarrow \texttt{0x09} &\Rightarrow \texttt{0xF6} \rightsquigarrow \texttt{0x09} \rightsquigarrow \texttt{0x5C}\\
\texttt{In} \leftarrow \texttt{0x0A} &\Rightarrow \texttt{0x5C} \rightsquigarrow \texttt{0x0A} \rightsquigarrow \texttt{0x26}\\
\texttt{In} \leftarrow \texttt{0x0B} &\Rightarrow \texttt{0x26} \rightsquigarrow \texttt{0x0B} \rightsquigarrow \texttt{0xFA}
\end{aligned}$$

Note that the identified code "happens to" allow the verifier to inspect *with absolute certainty* the platform's first 256 bytes. The rest is clear. The verifier does a last time measurement, allowing the Quine to print the device's first 256 bytes (power-off as soon as the last `bne` iteration completes, to avoid falling into the jaws of Aliens hiding beyond address `0x000B`).

It remains to check the Quine's payload (code between `0x000C` and `0x00FF`) and unleash the Quine's execution beyond address `0x000B`. Quine won the game.

7 Questions

This work raises a number of intriguing questions: Is it possible to prove security using only space constraints? In the negative, can we modify the assembly language to allow such proofs[10]? Can space-constrained Quines solve space-complete problems to flood memory instead of receiving random data?

Another interesting challenge consists in developing a time-constrained Quine whose proof does not require rebooting but the observation of one long succession of transitions. We conjecture that such programs exist. A possible starting point might be a code (not necessarily located at `start`) similar to:

```
loop: sta  Out
      lda  In
      sta  Out
      ldx  In
      stx  Out
      lda  ,X
      sta  Out
      bne  loop
```

Here the idea is that the verifier will feed the Quine with values chosen randomly in a specific set (to rule-out `dir`-variants) to repeatedly explore the code's immediate environment until some degree of certainty is acquired[11].

If possible, this would have the advantage of making the Quine a *function* automatically insertable into any application whose code needs to be authenticated. Moreover, if we manage to constrain the capabilities of such a Quine, *e.g.* not allow it read data beyond a given offset[12], we could offer the *selective* ability to audit critical program parts while preserving the privacy of others. For instance, the code of an accounting program could be audited while secret signature keys would provably remain out of the Quine's reach.

Finally, as time-constrained phenotyping is extremely quick (a few clock cycles), preserves nearly all the platform's data and requires only table lookups and comparisons, we currently try to extend the approach to more complex microprocessors and implement it between chips in motherboards.

[10] The approach would analogous to Java bytecode which is *purposely* shaped to fit type-inference.

[11] To exit the `bne` loop the verifier will purposely read a zero somewhere.

[12] *e.g.* the example above cannot read data beyond address 255.

References

1. J. Burger, D. Brill and F. Machi, *Self-reproducing programs*, Byte, volume 5, August 1980, pp. 74–75.
2. D. Hofstadter, *Godel, Escher, and Bach: An eternal golden braid*, Basic Books, Inc. New York, pp. 498–504.
3. Motorola Inc., 68HC(7)05H12 *General release specifications*, HC05H12GRS/D Rev. 1.0, November 1998.
4. T. Zeller, *The ghost in the* CD*; Sony* BMG *stirs a debate over software used to guard content*, The New York Times, C1, November 14, 2005.