Specification and Enforcement of Information Erasure Policies

Sebastian Hunt[1] David Sands[2] Filippo Del Tedesco[2]

[1]City University London

[2]Chalmers University of Technology, Sweden

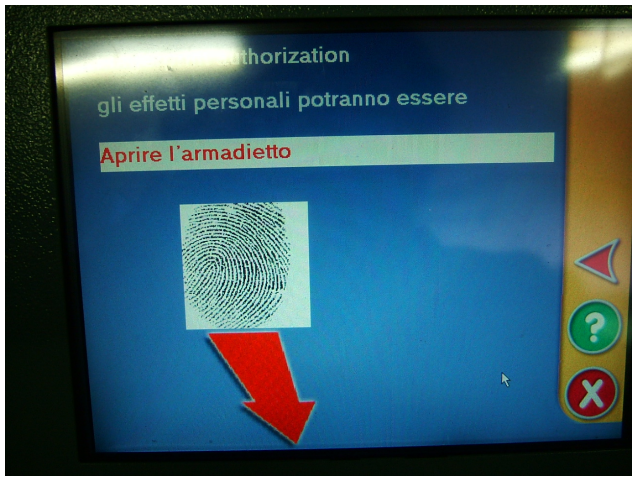
The 12th CREST Open Workshop
Security and Code
April 2011

Outline

Outline

Why Erase?

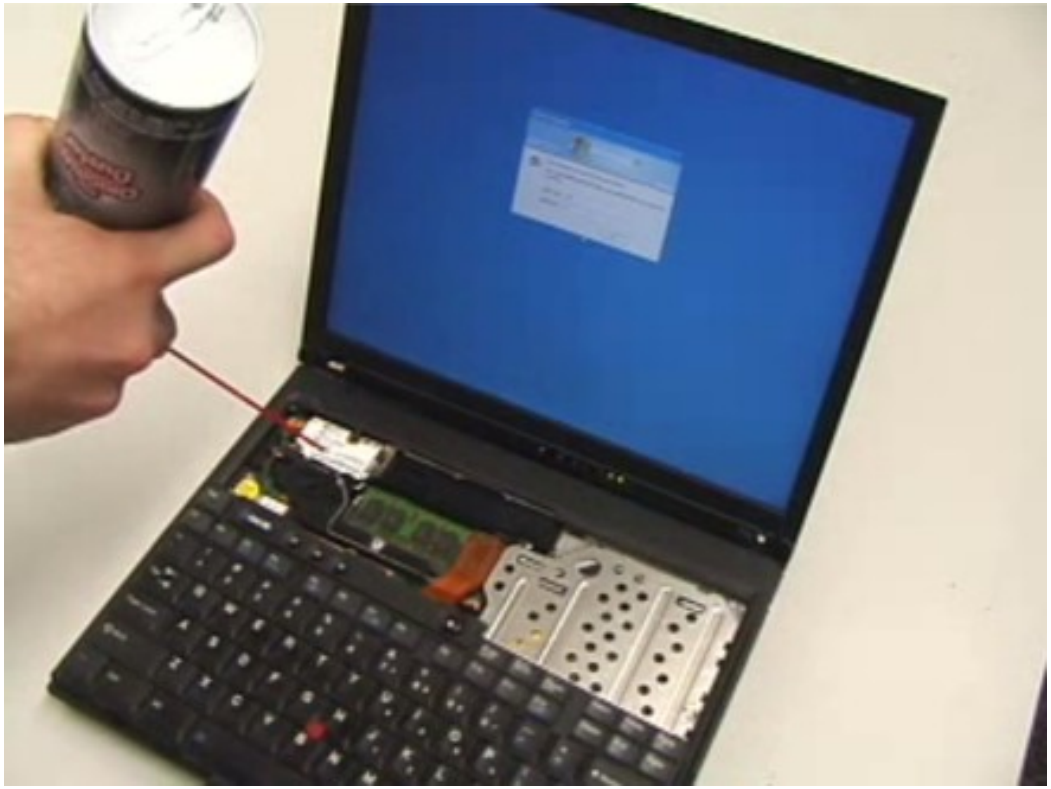
- Erasing sensitive information as soon as possible may give stronger security guarantees:
  - Erasure of identification tokens in e-voting.
  - Erasure of credit card details on completion of transaction.
  - Erasure of fingerprint data from a left-luggage locker.



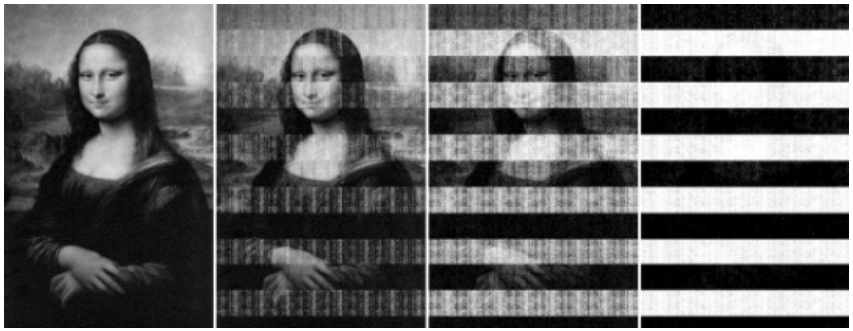
End-to-End Information Flow Policies

- We can't guarantee to erase information if we don't know where it has gone.
- Access control is not enough: we need end-to-end flow policies and enforcement.
- Control of information-flow is necessary. . . is it sufficient?
- If we can guarantee that information doesn't flow out of a system, is there any need to erase it?

Keeping Your Memory Clean



Cold Boot Attacks



- DRAM memory reverts to “ground state” when power is disconnected, but this takes several seconds.
  - even longer if memory modules are cooled
- Sensitive information residing in RAM can be stolen by cutting power and quickly rebooting into a custom kernel.
- [Halderman et al, 2008] recover disk-encryption keys
  - recover even partially degraded keys by exploiting precomputations kept in RAM by encryption software

Cold Boot Attack Countermeasures

- No silver bullet: keys must be stored *somewhere*.
- [Halderman et al, 2008]:

  > *"Countermeasures begin with efforts to avoid storing keys in memory. Software should overwrite keys when they are no longer needed, and it should attempt to prevent keys from being paged to disk."*

- Specific examples:
  - Avoid precomputation; erase precomputed data if unused for a given time interval
  - Obfuscate keys in RAM; recreate usable keys as necessary, erase ASAP

Outline

Erasure Policies

- Example: credit card number should be erased after transaction
- To erase contents of a variable, overwrite with a constant:

```
input cc from user;
t := transaction(cc);
output t to bank;
cc := 0
```

- Not enough to overwrite `cc`, must control flows *from* `cc`
- $\Rightarrow$ Can use a flow-sensitive security type system.

Erasure Policies

- Example: credit card number should be erased after transaction
- To erase contents of a variable, overwrite with a constant:

```
input cc from user;
t := transaction(cc);
output t to bank;
cc := 0
```

- Not enough to overwrite `cc`, must control flows *from* `cc`
- $\Rightarrow$ Can use a flow-sensitive security type system.

Erasure Policies

- Example: credit card number should be erased after transaction
- To erase contents of a variable, overwrite with a constant:

```
input cc from user;
t := transaction(cc);
output t to bank;
cc := 0
```

- Not enough to overwrite `cc`, must control flows *from* `cc`
- $\Rightarrow$ Can use a flow-sensitive security type system.

Flow-sensitive Security Type System

- Typing judgements: $\vdash \Gamma \, \{C\} \, \Gamma'$
  - $\Gamma, \Gamma' : \mathit{Var} \to \mathcal{L}$ (lattice of security levels)
  - variable types given by Γ on entry to C, by Γ' on exit

$$\frac{}{\vdash \Gamma \, \{x := E\} \, \Gamma[x \mapsto \Gamma(\mathbf{pc}) \sqcup \Gamma(E)]}$$

$$\frac{p = \Gamma(\mathbf{pc}) \quad \Gamma' = \Gamma[\mathbf{pc} \mapsto p \sqcup \Gamma(E)] \quad \vdash \Gamma' \, \{C\} \, \Gamma'}{\vdash \Gamma \, \{\mathbf{while}\ E\ C\} \, \Gamma}$$

Flow-sensitive Security Type System

- Typing judgements: $\vdash \Gamma \{C\} \Gamma'$
  - $\Gamma, \Gamma' : \mathit{Var} \to \mathcal{L}$ (lattice of security levels)
  - variable types given by Γ on entry to C, by Γ' on exit

$$\frac{}{\vdash \Gamma \{x := E\} \Gamma[x \mapsto \Gamma(\mathbf{pc}) \sqcup \Gamma(E)]}$$

$$\frac{p = \Gamma(\mathbf{pc}) \quad \Gamma' = \Gamma[\mathbf{pc} \mapsto p \sqcup \Gamma(E)] \quad \vdash \Gamma' \{C\} \Gamma'}{\vdash \Gamma \{\mathbf{while}\ E\ C\} \Gamma}$$

Flow-sensitive Security Type System

- Typing judgements: $\vdash \Gamma \, \{C\} \, \Gamma'$
  - $\Gamma, \Gamma' : \mathit{Var} \to \mathcal{L}$ (lattice of security levels)
  - variable types given by Γ on entry to C, by Γ' on exit

$$\frac{}{\vdash \Gamma \, \{x := E\} \, \Gamma[x \mapsto \Gamma(\mathbf{pc}) \sqcup \Gamma(E)]}$$

$$\frac{p = \Gamma(\mathbf{pc}) \quad \Gamma' = \Gamma[\mathbf{pc} \mapsto p \sqcup \Gamma(E)] \quad \vdash \Gamma' \, \{C\} \, \Gamma'}{\vdash \Gamma \, \{\mathbf{while}\ E\ C\} \, \Gamma}$$

Block-Structured Erase

- Basic idea: input a value x from channel i; use freely within command C; erase x (and all information derived from x) by the time C completes
- Generalise: erase *to some level a*

input $x : i$ **erase to** a **after** C

  - Allows information to be used after C but only at the higher security level a
  - Complete erasure captured by choosing $a = \#$: a security level higher than allowed for any variable

An Erasure Type System

$$\frac{\vdash \Gamma[x \mapsto p \sqcup \Gamma(i)]\ \{C\}\ \Gamma_1 \qquad \vdash \Gamma[x \mapsto p \sqcup a]\ \{C\}\ \Gamma_2 \qquad p = \Gamma(\mathbf{pc}) \qquad \Gamma'(y) = \begin{cases} \Gamma_1(y) & \text{if } y \in \mathit{OVar} \\ \Gamma_1(y) \sqcup \Gamma_2(y) & \text{otherwise} \end{cases}}{\vdash \Gamma\ \{\mathbf{input}\ x : i\ \mathbf{erase\ to}\ a\ \mathbf{after}\ C\}\ \Gamma'[i \mathrel{+}= p]}$$

- environments track flows between variables and on IO channels ($i \in \mathit{IVar}, o \in \mathit{OVar}$)
- two typings for body: one for flows inside body, one for flows to rest of program

An Erasure Type System

$$\frac{\vdash \Gamma[x \mapsto p \sqcup \Gamma(i)]\ \{C\}\ \Gamma_1 \qquad \vdash \Gamma[x \mapsto p \sqcup a]\ \{C\}\ \Gamma_2 \qquad p = \Gamma(\mathbf{pc}) \qquad \Gamma'(y) = \begin{cases} \Gamma_1(y) & \text{if } y \in \mathit{OVar} \\ \Gamma_1(y) \sqcup \Gamma_2(y) & \text{otherwise} \end{cases}}{\vdash \Gamma\ \{\mathbf{input}\ x : i\ \mathbf{erase\ to}\ a\ \mathbf{after}\ C\}\ \Gamma'[i \mathrel{+}= p]}$$

- environments track flows between variables and on IO channels ($i \in \mathit{IVar}, o \in \mathit{OVar}$)
- two typings for body: one for flows inside body, one for flows to rest of program

Outline

An Implementation Problem (Solved)

Typing is Exponential

$$\frac{\vdash \Gamma[x \mapsto p \sqcup \Gamma(i)]\ \{C\}\ \Gamma_1 \qquad \vdash \Gamma[x \mapsto p \sqcup a]\ \{C\}\ \Gamma_2 \qquad p = \Gamma(\mathbf{pc}) \qquad \Gamma'(y) = \begin{cases} \Gamma_1(y) & \text{if } y \in \mathit{OVar} \\ \Gamma_1(y) \sqcup \Gamma_2(y) & \text{otherwise} \end{cases}}{\vdash \Gamma\ \{\mathbf{input}\ x : i\ \mathbf{erase\ to}\ a\ \mathbf{after}\ C\}\ \Gamma'[i \mathrel{+}= p]}$$

- Double typing required for C.
- Nesting an **input** command inside C doubles the size of the derivation.
- $\Rightarrow$ Typing is exponential in the size of the program.

An Implementation Problem (Solved)

Typing is Exponential

$$\frac{\vdash \Gamma[x \mapsto p \sqcup \Gamma(i)]\ \{C\}\ \Gamma_1 \qquad \vdash \Gamma[x \mapsto p \sqcup a]\ \{C\}\ \Gamma_2 \qquad p = \Gamma(\mathbf{pc}) \qquad \Gamma'(y) = \begin{cases} \Gamma_1(y) & \text{if } y \in \mathit{OVar} \\ \Gamma_1(y) \sqcup \Gamma_2(y) & \text{otherwise} \end{cases}}{\vdash \Gamma\ \{\textbf{input}\ x : i\ \textbf{erase to}\ a\ \textbf{after}\ C\}\ \Gamma'[i \mathrel{+}= p]}$$

- Double typing required for C.
- Nesting an **input** command inside C doubles the size of the derivation.
- $\Rightarrow$ Typing is exponential in the size of the program.

An Implementation Problem (Solved)

Typing is Exponential

$$\frac{\vdash \Gamma[x \mapsto p \sqcup \Gamma(i)]\ \{C\}\ \Gamma_1 \qquad \vdash \Gamma[x \mapsto p \sqcup a]\ \{C\}\ \Gamma_2 \qquad p = \Gamma(\mathbf{pc}) \qquad \Gamma'(y) = \begin{cases} \Gamma_1(y) & \text{if } y \in \mathit{OVar} \\ \Gamma_1(y) \sqcup \Gamma_2(y) & \text{otherwise} \end{cases}}{\vdash \Gamma\ \{\textbf{input}\ x : i\ \textbf{erase to}\ a\ \textbf{after}\ C\}\ \Gamma'[i \mathrel{+}= p]}$$

- Double typing required for C.
- Nesting an **input** command inside C doubles the size of the derivation.
- $\Rightarrow$ Typing is exponential in the size of the program.

Solution: Principal Typing

$$\frac{\vdash C : \Delta \qquad \Delta_j = \Delta \,;\, \eta[x \mapsto \{\mathbf{pc}, i_j\}] \qquad j = 1, 2 \qquad \Delta'(x) = \begin{cases} \Delta_1(x) & \text{if } x \in \mathit{OVar} \\ \Delta_1(x) \cup \Delta_2(x) & \text{otherwise} \end{cases}}{\vdash \textbf{input } x : i_1 \textbf{ erase to } i_2 \textbf{ after } C : \Delta'[i_1 \mathrel{+}= \{\mathbf{pc}\}]}$$

- (Note: “erased to” type now a variable.)
- Transformed type system which derives principal type Δ
  - all other typings can be *calculated* from Δ
- Derivation size linear in size of $C \Rightarrow$ polynomial typing
  - also extends to polynomial typing for procedures

Type System Too Weak; Erasure Condition Too Strong

Example: Fingerprint Locker

```
input fp : reader erase to # after {
    locked := true;
    while (locked) {
        input fpTry : reader erase to # after {
            locked := !match(fpTry, fp);
            fpTry := "";
        }
    }
    fp := "";
}
```

- Outer input is erasing; `locked` is always false on exit but type system is too weak to verify this
- Inner input is *not* erasing; variation in `locked`, but we need to allow this

Type System Too Weak; Erasure Condition Too Strong

Example: Fingerprint Locker

```
input fp : reader erase to # after {
    locked := true;
    while (locked) {
        input fpTry : reader erase to # after {
            locked := !match(fpTry, fp);
            fpTry := "";
        }
    }
    fp := "";
}
```

- Outer input is erasing; `locked` is always false on exit but type system is too weak to verify this
- Inner input is *not* erasing; variation in `locked`, but we need to allow this

Type System Too Weak; Erasure Condition Too Strong

Example: Fingerprint Locker

```
input fp : reader erase to # after {
    locked := true;
    while (locked) {
        input fpTry : reader erase to # after {
            locked := !match(fpTry, fp);
            fpTry := "";
        }
    }
    fp := "";
}
```

- Outer input is erasing; `locked` is always false on exit but type system is too weak to verify this
- Inner input is *not* erasing; variation in `locked`, but we need to allow this

Semantic Model Not Sufficiently Expressive

Example: Special Offer Code

- Soundness proof for type system based on a *stream model* of the environment which cannot capture what is wrong with this example:

```
input cc :  user erase to # after {
    output specialOffer(cc) to user;
    t := transaction(cc);
    output t to bank;
    t := 0;
    cc := 0;
}
```

Semantic Model Not Sufficiently Expressive

Example: Special Offer Code

- Soundness proof for type system based on a *stream model* of the environment which cannot capture what is wrong with this example:

```
input cc :  user erase to # after {
    output specialOffer(cc) to user;
    t := transaction(cc);
    output t to bank;
    t := 0;
    cc := 0;
}

input special from user;
cc := decrypt(special)
```

Summary

- Verifiable erasure guarantees are an essential aspect of security.
- Simple forms of erasure policy can be enforced at language level by type systems.
  - Care needed to avoid exponential blow-up.
- Richer semantic models needed to formalise more flexible erasure policies.
- Smarter type system / theorem proving needed to verify them.