Decompilation is an information-flow problem

(Or, information flow meets program transformation)

Boris Feigin

Computer Laboratory, University of Cambridge

PLID 2008

joint work with Alan Mycroft

Motivation

"Given suitable tools we can present the [cryptographic] key as a constant in the computation which is carried out using that key and then we can optimise the code given that constant. This will cause the key to be intimately intertwined with the code which uses it."

Playing 'Hide and Seek' with Stored Keys
Shamir and van Someren (1999)

Typical source and target languages

$$\begin{array}{lcl} v & \in & \mathsf{Value} = \mathbb{Z} \\ r & \in & \mathsf{Register} = \{\mathtt{r0}, \mathtt{r1}, \ldots, \mathtt{r31}\} \end{array}$$

`while`-language (source):

$$\begin{array}{lcl} e & ::= & v \mid x \mid op\ e_1, \ldots, e_n \\ c & ::= & x := e \mid \mathtt{skip} \mid c_0;\ c_1 \\ & \mid & \mathtt{if}\ e\ \mathtt{then}\ c_0\ \mathtt{else}\ c_1 \mid \mathtt{while}\ e\ \mathtt{do}\ c \end{array}$$

RISC assembly (target):

$$\begin{array}{lcl} \iota & ::= & \mathtt{movi}\ r_d,\ v \mid \mathtt{mov}\ r_d,\ r_s \mid \mathtt{ld}\ r_d,\ [r_s] \mid \mathtt{st}\ [r_d],\ r_s \\ & \mid & op\ r_d,\ r_1, \ldots, r_n \mid \mathtt{jz}\ r,\ l \mid \mathtt{jnz}\ r,\ l \mid \mathtt{nop} \mid \iota_0;\ \iota_1 \end{array}$$

Definitions

- $\mathcal{C}(-)$ is a compiler from source language S to target language T.
- The observational equivalence relations of S and T are (respectively) $\sim_S$ and $\sim_T$.
- Decompilation recovers a source program semantically equivalent to the original. $\mathcal{D}(-)$ is a decompiler iff

$$\mathcal{D}(\mathcal{C}(e)) \in [e]_{\sim_S}$$

  This is the weakest possible definition of decompilation.
    - In certain cases there is a trivial solution for $\mathcal{D}(-)$: emit an interpreter for T written in S incorporating the text of the program (in T) to be decompiled.

Definitions

- $\mathcal{C}(-)$ is a compiler from source language S to target language T.
- The observational equivalence relations of S and T are (respectively) $\sim_S$ and $\sim_T$.
- Decompilation recovers a source program semantically equivalent to the original. $\mathcal{D}(-)$ is a decompiler iff

  $$\mathcal{D}(\mathcal{C}(e)) \in [e]_{\sim_S}$$

  This is the weakest possible definition of decompilation.
  - In certain cases there is a trivial solution for $\mathcal{D}(-)$: emit an interpreter for T written in S incorporating the text of the program (in T) to be decompiled.
- How well can a decompiler do in principle?
  - IOW, how much *information* about the source program can be inferred from the output of the compiler?

Example

$$\mathcal{C}(\text{“}x := 42\text{”}) = \mathcal{C}(\text{“}y := 42;\ \mathsf{x} := \mathsf{y}\text{”}) =$$

$$= \mathcal{C}(\text{“}z := 6;\ \ y := 7;\ \ x := z \times y\text{”}) =$$

$$= \text{“}\texttt{mov r0},\ 42\text{”}$$

$\mathcal{C}(-)$ does constant folding, constant propagation, etc.

Program equivalence

- $\equiv$ ("bit-for-bit" equality of programs)

$$e \equiv e' \iff \textbf{strcmp}(e, e') == 0$$

- $\sim_\alpha$ (α-equivalence)

Program equivalence

- Recall: two expressions are contextually equivalent ($e \sim e'$) whenever

  $$e \sim e' \iff \forall \mathrm{Ctx}[-] \quad \mathrm{Ctx}[e] \cong \mathrm{Ctx}[e']$$

  where $\mathrm{Ctx}[-]$ ranges over contexts of the language and $\cong$ is some observation (say, convergence).

Program equivalence

- Recall: two expressions are contextually equivalent ($e \sim e'$) whenever

$$e \sim e' \iff \forall \mathrm{Ctx}[-] \quad \mathrm{Ctx}[e] \cong \mathrm{Ctx}[e']$$

  where $\mathrm{Ctx}[-]$ ranges over contexts of the language and $\cong$ is some observation (say, convergence).
- Restriction to programs (d ranges over inputs):

$$e \sim e' \iff \forall d \in D \quad [\![e]\!](d) = [\![e']\!](d)$$

Example: `size_t strlen(const char *str)`

```c
const char *s = str;

while(*s)
        s++;
return (s - str);
```

```c
size_t len = 0;

for(; str[len]; len++)
        ;
return len;
```

Intuition

Define the relation $f^{-1}(Q)$, the kernel of f w.r.t. Q (Clark et al., 2005):

$$x\ f^{-1}(Q)\ x' \iff (f\ x)\ Q\ (f\ x')$$

Intuition

Define the relation $f^{-1}(Q)$, the kernel of f w.r.t. Q (Clark et al., 2005):

$$x \; f^{-1}(Q) \; x' \iff (f \; x) \; Q \; (f \; x')$$

E.g.

$$\text{“}x := 42\text{”} \quad \mathcal{C}^{-1}(\equiv) \quad \text{“}y := 42;\ \mathrm{x} := \mathrm{y}\text{”}$$

Intuition

Define the relation $f^{-1}(Q)$, the kernel of f w.r.t. Q (Clark et al., 2005):

$$x\ f^{-1}(Q)\ x' \iff (f\ x)\ Q\ (f\ x')$$

E.g.

$$\text{“}x := 42\text{”} \quad \mathcal{C}^{-1}(\equiv) \quad \text{“}y := 42;\ x := y\text{”}$$

Programs compiled by “less normalizing” compilers are more susceptible to decompilation. We tend to have the case that:

$$\sim_\alpha \subset \mathcal{C}_1^{-1}(\equiv) \subset \mathcal{C}_2^{-1}(\equiv) \subset \ldots \subset \mathcal{C}_n^{-1}(\equiv) \subset \sim_S$$

where $\mathcal{C}_1(-)$ to $\mathcal{C}_n(-)$ are progressively more optimizing compilers.

Compiler correctness

$\mathcal{C}(-)$ is fully abstract (Abadi, 1998) iff

$$e \sim_S e' \iff \mathcal{C}(e) \sim_T \mathcal{C}(e') \tag{1}$$

Compiler correctness

$\mathcal{C}(-)$ is fully abstract (Abadi, 1998) iff

$$e \sim_S e' \iff \mathcal{C}(e) \sim_T \mathcal{C}(e') \tag{1}$$

Abadi observes that the forward implication "means that the translation does not introduce information leaks".

Non-interference

$$e \sim_S e' \;\Rightarrow\; \mathcal{C}(e) \sim_T \mathcal{C}(e') \tag{2}$$

Zero information flow (from high-security inputs to low-security outputs) for a program M:

$$\sigma \sim_{\mathrm{low}} \sigma' \Rightarrow [\![M]\!](\sigma) \approx [\![M]\!](\sigma') \tag{3}$$

where two states are equivalent up to $\sim_{\mathrm{low}}$ when their low-security parts are equal.

Relating non-interference and software protection

Let P and Q be binary relations over domains D and E respectively. Then, given $f : D \to E$, say that $f : P \Rightarrow Q$ whenever

$$\forall x, x' \in D \qquad x\ P\ x' \Rightarrow (f\ x)\ Q\ (f\ x')$$

Relating non-interference and software protection

Let P and Q be binary relations over domains D and E respectively. Then, given $f : D \to E$, say that $f : P \Rightarrow Q$ whenever

$$\forall x, x' \in D \qquad x \; P \; x' \Rightarrow (f \; x) \; Q \; (f \; x')$$

The correspondence is explicit:

$$[\![M]\!](-) \; : \sim_{\mathrm{low}} \Rightarrow \approx \qquad\qquad \mathcal{C}(-) \; : \sim_S \Rightarrow \sim_T$$

The substitution $\{\mathcal{C} \;/\; [\![M]\!], \; \sim_S \;/\; \sim_{\mathrm{low}}, \; \sim_T \;/\; \approx\}$ unifies the equations nicely.

Parallels

- Programs are secret (high-security) inputs. Compiled binaries are the public (low-security) outputs ($\equiv$).
- Attackers attempt to infer (as much as possible about) the inputs from the outputs. (Decompilation.)

Parallels

- Programs are secret (high-security) inputs. Compiled binaries are the public (low-security) outputs ($\equiv$).
- Attackers attempt to infer (as much as possible about) the inputs from the outputs. (Decompilation.)

Caveat: in practice, the goal of decompilation is to recover *any* readable source program.

Secure information flow for compilers?

We would like to have zero information flow compilers:

$$\mathcal{C}(-) \ : \sim_S \Rightarrow \equiv$$

Secure information flow for compilers?

We would like to have zero information flow compilers:

$$\mathcal{C}(-)\ :\sim_S \Rightarrow \equiv$$

- Relational reading: $\mathcal{C}(-)$ may leak only the equivalence class of its input programs.
- $\mathcal{C}(-)$ must be perfectly optimizing (undecidable for Turing-complete languages).
  - Though, cf. superoptimization (Massalin, 1987).

Implications

In general, a compiler must leak more than just the equivalence class of its input programs. We are interested in applying techniques from quantitative information flow to deriving concrete bounds on the leakage.

E.g.: the identity “compiler” ($\lambda x.x$) leaks its input completely.

Possible applications

- Randomized compilation and information-flow security for non-deterministic languages
  - cf. non-deterministic encryption schemes
- Obfuscation (more generally: software protection)

Virtualization

Essentially, fast whole-system emulation. Examples: KVM, VMware, Xen, . . .

Virtualization

Essentially, fast whole-system emulation. Examples: KVM, VMware, Xen, . . .

(virtual machine) **transparency n.**

> *making virtual and native hardware indistinguishable under close scrutiny by a dedicated adversary*
>
> *(Garfinkel et al., 2007)*

Virtualization

Essentially, fast whole-system emulation. Examples: KVM, VMware, Xen, . . .

(virtual machine) **transparency n.**

> *making virtual and native hardware indistinguishable under close scrutiny by a dedicated adversary*
>
> *(Garfinkel et al., 2007)*

$$e \sim_{x86} e' \iff [\![vm]\!](e) \approx [\![vm]\!](e')$$

From compilers to interpreters and back again

- Partial evaluation

$$[\![e]\!](d) = [\![sint]\!](e, d) = [\![[\![mix]\!](sint, e)]\!](d)$$

From compilers to interpreters and back again

- Partial evaluation

$$[\![e]\!](d) = [\![sint]\!](e, d) = [\![[\![mix]\!](sint, e)]\!](d)$$

- Non-interference?

$$e \sim_S e' \quad \Longleftrightarrow \quad \forall d \quad [\![int]\!](e, d) \quad \approx \quad [\![int]\!](e', d)$$

$$e \sim_S e' \quad \Longleftrightarrow \quad [\![mix]\!](int, e) \quad \approx \quad [\![mix]\!](int, e')$$

Overview

- Optimizing compilers obey a “non-interference”-like property
- Perfect optimization is impossible, so information leaks are inevitable
- An information-flow approach to program transformation?

Challenges

- Probability distributions over programs
  - Shannon information theory / Kolmogorov complexity / Scott's information systems
- "Real" compilers don't come with formalized equational theories

Related work

- Decompilation: Mycroft (1999), Katsumata and Ohori (2001), Ager et al. (2002).
- Full abstraction: Mitchell (1993), Abadi (1998), Kennedy (2006).
- Reverse engineering by power analysis etc.: Vermoen (2007).
- Randomized compilation: Cohen (1993), Forrest et al. (1997).
- Nullspace of compilers: Veldhuizen and Lumsdaine (2002).
- Obfuscation: Barak et al. (2001), Dalla Preda and Giacobazzi (2005).
- Virtual machines and partial evaluation: Feigin and Mycroft (2008).

Questions?