Language-Based Security

Notes from OPLSS@BU 2024, Lecture 4

Lecturer: Stephen Chong
Scribe: Elanor Tang

June 13, 2024

1 Enforcing Language Abstractions

Language abstractions hide details of execution and memory layout, allowing us to reason about program security. In practice, these abstractions can be violated—one example is compilation which composes with foreign code that violates language abstractions. Moreover, languages can have strange features: reflection, unsafe blocks of code, dynamic code that evaluates a string as a program, etc., can all break abstractions. Thus, we need methods for enforcing language abstractions, which encode security guarantees, during execution:

- No executable data: prevent execution of unauthorized code. For example, you could exclude the `eval` operator from a language. It may be useful to allow limited forms of reflection, such as allowing a program to manipulate the program or create new pieces of executable content.
- Fat pointers: using pointers enhanced with upper/lower bounds on the available region may prevent buffer overflows.
- Software fault isolation (SFI): limit what code can execute and what memory can be accessed. One example is control flow integrity (CFI), which restricts the addresses execution can jump to.

These approaches do not enforce the semantics of the original language, but merely reduce the possible behaviors to an approximation of the original source program. For example, CFI may only allow a program to jump to a valid target, but perhaps it could still do so at a time outside the control flow of the original program. Thus, we want to actually guarantee enforcement of language semantics.

1.1 Secure Compilation

Compilation frequently results in target code which violates language semantics of the source code. Since semantics are a method of enforcing security properties, this can create security vulnerabilities. Thus, we want **secure compilation:** security properties at the source level should still be valid after compilation.

1.2 Contextual Equivalence

A **context** $\mathbb{C}$ is a program with a hole $[\cdot]$; this can be filled with a program P to generate a whole program $\mathbb{C}[P]$. Contexts can model situations like linking, where a program is imported from another file, and the hole is the import statement.

We write $P \Downarrow o$ if a (whole) program P produces observation o; this observation can be divergence or termination with output. Two components P_1 and P_2 are **contextually equivalent**, notated $P_1 \simeq_{\text{ctx}} P_2$, if

$$\forall \mathbb{C}, o.\ \mathbb{C}[P_1] \Downarrow o \iff \mathbb{C}[P_2] \Downarrow o.$$

The idea is that two programs are equivalent if no context can be used to distinguish between them—we will use this at the levels of both source and target programs. Notably, two programs that assign different values to a secret value may be considered contextually equivalent, since the output appears the same.

Contextual equivalence, however, fails to capture the time taken for an execution to terminate.

1.3 Full Abstraction

A compiler is **fully abstract** if

$$\forall P_1, P_2.\ P_1 \simeq_{\text{ctx}} P_2 \iff [\![P_1]\!] \simeq_{\text{ctx}} [\![P_2]\!].$$

Here, $[\![P]\!]$ denotes the compilation of a program P.

We call the backward direction **reflection** of contextual equivalence, and it follows from compiler correctness. (A very high-level definition of compiler correctness is that target-level behaviors are a subset of the possible behaviors of source-level code. That is, a compiler does not introduce any new behaviors.) For example, an incorrect compiler which compiles every program to "return 42" would not satisfy reflection.

We call the forward direction **preservation** of contextual equivalence; it implies the target language cannot make any additional distinctions between P_1 and P_2. That is, there is no additional ability to break source-level abstractions.

Achieving full abstraction may require **back translation**: proving any target-language context can be expressed as a source-language context. This can be used to prove preservation.

Static techniques include using a type system to restrict target language contexts to those translatable back to the source language. Dynamic techniques including using homomorphic encryption on some target values to limit operations on the encrypted data, inserting runtime checks, or using security architectures such as address space layout randomization, or Trusted Execution Environments.

Full abstraction is hard to achieve in practice: the downside of full abstraction would be losing current compilation infrastructure and efficiency.

1.4 Beyond Full Abstraction

While full abstraction does preserve and reflect contextual equivalence, we also care about other properties: safety, liveness, and hyperproperties (properties over sets of

execution traces) such as noninterference. Full abstraction is not strong enough to enforce hyperproperties, and conversely, it may be too hard to enforce if we only care about safety.

For example, consider a compiler that translates programs of the form

```
f(x:Bool) ↦ e
```

to

```
f(x:Nat) ↦ if x < 2 then e↓ else x < 3 then f(x) else 42
```

where $e\downarrow$ is the compilation of e.

Here a boolean input is compiled to a natural number, with expected behavior on $\{0, 1\}$ and new behavior on all other inputs. This compiler is fully abstract: provided there is no context that can distinguish between programs in the source language, there is no context that can distinguish between programs in the target language, and vice versa. But it clearly does not satisfy the safety property of "Never output 42."

1.5 Robust Preservation

We provide several definitions of secure compilation which go beyond full abstraction.

1.5.1 Robust Trace Preservation (RTP)

A compiler satisfies RTP if and only if compilation preserves every trace-based property:

$$\forall \pi \in 2^{Trace}.\ \forall P.\ (\forall \mathbb{C}_S, t.\ \mathbb{C}_S[P] \rightsquigarrow t \Rightarrow t \in \pi) \Rightarrow (\forall \mathbb{C}_T, t.\ \mathbb{C}_T[P\downarrow\ ] \rightsquigarrow t \Rightarrow t \in \pi).$$

Here, π is a trace-based property in the set of all traces, and $t \in \pi$ means t is a trace satisfying π. Blue corresponds to source contexts and programs, while orange corresponds to target contexts. We read the $\mathbb{C}[P] \rightsquigarrow t$ as producing the trace t. Note that here we gloss over relating source and target traces (in practice they may look different).

Equivalently, we can define this without referring to an explicit property $\pi \in 2^{Trace}$.

$$\forall P.\ \forall \mathbb{C}_T.\ \forall t.\ \mathbb{C}_T[P\downarrow\ ] \rightsquigarrow t \Rightarrow \exists \mathbb{C}_S.\ \mathbb{C}_S[P] \rightsquigarrow t.$$

1.5.2 Robust Safety Preservation (RSP)

We specialize this definition to safety: a compiler satisfies RSP if and only if compilation preserves every trace-based safety property:

$$\forall \pi \in Safety.\ \forall P.\ (\forall \mathbb{C}_S, t.\ \mathbb{C}_S[P] \rightsquigarrow t \Rightarrow t \in \pi) \Rightarrow (\forall \mathbb{C}_T, t.\ \mathbb{C}_T[P\downarrow\ ] \rightsquigarrow t \Rightarrow t \in \pi).$$

The "property-free" definition is

$$\forall P.\ \forall \mathbb{C}_T.\ \forall m.\ \mathbb{C}_T[P\downarrow\ ] \rightsquigarrow m \Rightarrow \exists \mathbb{C}_S.\ \mathbb{C}_S[P] \rightsquigarrow m$$

where m is a finite trace prefix; we can think of it as a "bad" trace that breaks the safety property. In other words, if the safety property is broken in a target context, it is broken in some source context; by the contrapositive, if the safety property is upheld in all target contexts, it is upheld in all source contexts.

1.5.3 Robust Hyperproperty Preservation (RHP)

We can generalize these definitions to hyperproperties: a compiler satisfies RHP if and only if compilation preserves every trace-based hyperproperty:

$$\forall H \in 2^{Trace^{Trace}}.\ \forall P.\ (\forall \mathbb{C}_S.\ Behav(\mathbb{C}_S[P]) \in H) \Rightarrow (\forall \mathbb{C}_T.\ Behav(\mathbb{C}_T[P\downarrow\,]) \in H).$$

Now, we have sets of hyperproperties, which themselves are sets of trace-based properties. In words, if the set of source behaviors is in the hyperproperty, then the set of target behaviors is in the hyperproperty.

Equivalently,

$$\forall P.\ \forall \mathbb{C}_T.\ \exists \mathbb{C}_S.\ Behav(\mathbb{C}_T[P\downarrow\,]) = Behav(\mathbb{C}_S[P])$$

or

$$\forall P.\ \forall \mathbb{C}_T.\ \exists \mathbb{C}_S.\ \forall t.\ \mathbb{C}_T[P\downarrow\,] \rightsquigarrow t \iff \mathbb{C}_S[P] \rightsquigarrow t.$$

1.5.4 Relating Secure Compilation Criteria

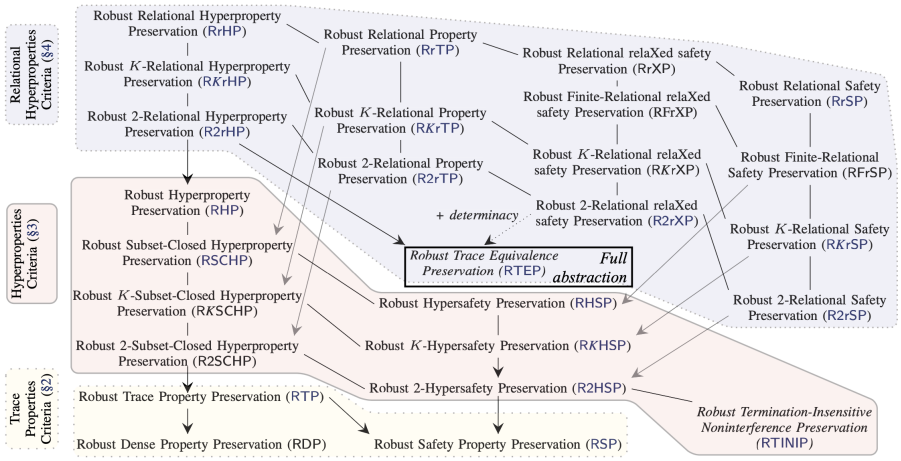


Fig. 1: Partial order with the secure compilation criteria studied in this paper. Criteria higher in the diagram imply the lower ones to which they are connected by edges. Criteria based on trace properties are grouped in a yellow area, those based on hyperproperties are in a red area, and those based on relational hyperproperties are in a blue area. Criteria with an *italics name* preserve a *single* property that belongs to the class they are connected to; the dotted edge requires an additional determinacy assumption. Finally, each edge with a thick arrow denotes a *strict* implication that we have proved as a separation result.

Abate et al., CSF 2019.

Observe that full abstraction is the same as robust trace equivalence preservation.

2 Weird Machines

There are several security attacks which exploit features of the stack and memory layout.

- Buffer overflow exploit: Consider a function that copies input to a buffer. An attacker can pass in data that overflows the provided buffer, overwriting the function's return address. Upon returning from the function, arbitrary code is executed.
- Return to libc attack: Rather than overwriting the return address with arbitrary code (which may not work if the stack is non-executable), an attack can overwrite it with the address of a system function in libc. If the attacker also overwrites the argument build area with the address of the string `/bin/sh`, then the program will jump to the system and open up a shell for the attacker.
- Return-Oriented Programming (ROP): ROP chains **gadgets**, short sequences of machine instructions which end in a return—each gadget performs a small amount of computation, then the return address jumps to the next gadget. An attacker can automatically identify gadgets and overwrite the stack with a chain of them to execute a desired computation.

Weird machines, based on a 2020 paper by Dullien, provide a model for formalizing and generalizing these kinds of vulnerabilities.

2.1 Intended Finite State Machines (IFSM)

IFSMs describe the intended implementation of the programmer. An IFSM is notated as $\Theta = (Q, iF, \Sigma, \Delta, \gamma, \sigma)$, with

- Set of states Q
- Initial state i
- Final states F
- Input alphabet Σ
- Output alphabet Δ
- State transition function $\delta : Q \times \Sigma \to Q$
- Output function $\sigma : Q \times \Sigma \to \Delta$

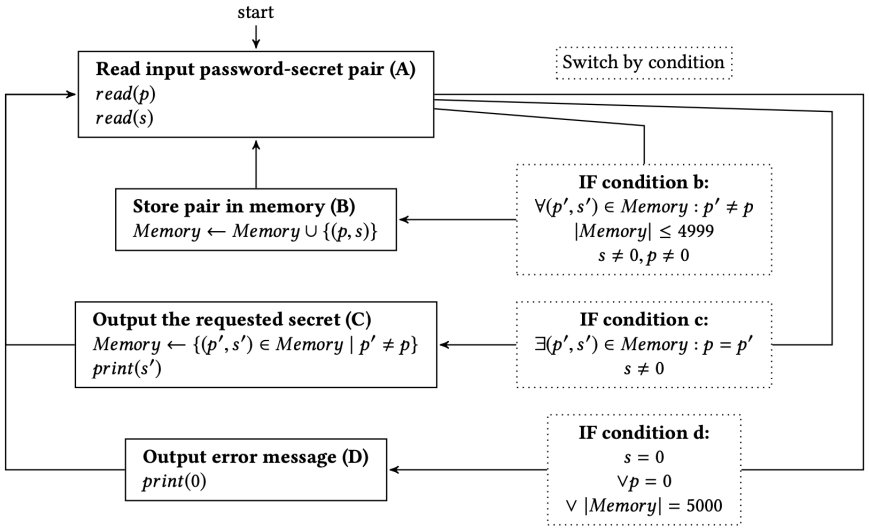
2.2 Example: Tiny Secure Message Passing Server

Consider a machine that remembers password-secret pairs for later retrieval which removes the pair. We set an arbitrary limit of 5000 password-secret pairs. Intuitively, we want security to mean needing to know (or guess) the right password to obtain the secret.

2.2.1 IFSM

We define the corresponding IFSM as follows:

$$\mathcal{M} := \left\{ \begin{array}{l} \emptyset, \\ \{(p_1, s_1)\}, \\ \ldots, \\ \{(p_1, s_1), \ldots, (p_{5000}, s_{5000})\} \end{array} \middle| \begin{array}{l} p_i, s_i \in bits_{32} \backslash \{0\} \\ p_i \neq p_j \end{array} \right\}$$



$$Q := \{A_M, M \in \mathcal{M}\}, \qquad i := A_\emptyset, F := \emptyset$$

$$\Sigma := \{(p, s) | p, s \in bits_{32}\}, \qquad \Delta := \{s \in bits_{32}\}$$

$$\delta := A_M \times (p, s) \quad \rightarrow \quad \begin{cases} A_{M \cup (p,s)} \text{ if } \begin{array}{l} (p, s) \notin M \\ \wedge |M| \leq 4999 \\ \wedge s \neq 0 \end{array} \\ A_{M \backslash (p,s)} \text{ if } (p, s) \in M \\ A_M \text{ otherwise} \end{cases}$$

$$\sigma := A_M \times (p, s) \quad \rightarrow \quad \begin{cases} s' \text{ if } (p, s') \in M \\ 0 \text{ if } s = 0 \vee |M| = 5000 \end{cases}$$

2.2.2 Security Property

To define the security property, we set up the following game:

(1) The attacker chooses a probability distribution $\mathcal{A}$ over finite-state transducers Θ_{exploit} that have an input alphabet $\Sigma_{\Theta_{\text{exploit}}} = \Delta$ and output alphabet $\Delta_{\Theta_{\text{exploit}}} = \Sigma$. This means that the attacker specifies one or more finite-state transducers that take as input the outputs of the IFSM, and output words that are the input for the IFSM.
(2) Once this is done, the defender draws two elements p, s from $bits_{32}$ according to the uniform distribution.
(3) The attacker draws a finite-state transducer from his distribution and is allowed to have it interact with the IFSM for an attacker-chosen number of steps n_{setup}.
(4) The defender sends his (p, s) to the IFSM.
(5) The attacker gets to have his Θ_{exploit} interact with the IFSM for a further attacker-chosen number of steps n_{exploit}.

We want the probability that $\Theta_{exploit}$ obtains the secret to be no better than guessing:

$$\Pr[s \in o_{IFSM}] \leq \frac{n_{setup} + n_{exploit}}{|bits_{32}|} = \frac{|o_{exploit}|}{2^{32}}.$$

2.2.3 Emulating the IFSM

We emulate the IFSM with a Cook-and-Reckhow RAM machine model that uses Harvard architecture (where the code is not data). We assume there are 2^{16} 32-bit memory cells, where the first 6 are treated as registers.

We consider two variants:

- Variant 1: Use cells 0-5 as scratch and cells 6-10006 as a simple flat array for storing pairs of values. Upon receiving a query, search through memory for empty pairs of memory cells.
- Variant 2: Rather than an array, use two singly-linked lists: one will track free space for password-secret pairs, and the other will track currently active password-secret pairs.

2.2.4 Sane, Transitory, and Weird States

We use the IFSM as the intensional specification, and call the implementation machine *cpu*. Let Q_{cpu} be the set of states of the implementation machine.

Let $\alpha_{\Theta,cpu,\rho} : Q_{cpu} \rightharpoonup Q_\Theta$ be the (partial) abstraction function from states Q_{cpu} to states Q_Θ of the IFSM. Denote Q_{cpu}^{sane} as the states for which $\alpha_{\Theta,cpu,\rho}$ is defined (which directly correspond to a state of the IFSM).

Since *cpu* may take multiple steps to implement one step in the IFSM, we also define transitory states Q_{cpu}^{trans} (to distinguish them from error states).

We can now define weird states $Q_{cpu}^{weird} = Q_{cpu} \setminus (Q_{cpu}^{sane} \cup Q_{cpu}^{trans})$, which consist of all remaining states. Formally, bugs happen when cpu reaches a state in Q_{cpu}^{weird}. In practice, weird states may come from human error (most common!), hardware faults, or transcription errors.

2.2.5 Formally Defining Weird Machines

The intended machine implementation is to emulate all state transitions of the IFSM with transitions between sane states which may involve transitory states.

$$(Q_{cpu}^{sane} \cup Q_{cpu}^{trans}, q_{init}, Q_{cpu}^{sane} \cup Q_{cpu}^{trans}, \Sigma, \Delta, \delta, \sigma)$$

A unintended, weird machine starts in a weird state and transitions to weird states, using "instructions" in the form of input—that is, transitions meant for transforming valid states.

$$(Q_{cpu}^{weird}, q_{init}, Q_{cpu}^{sane} \cup Q_{cpu}^{trans}, \Sigma', \Delta', \delta', \sigma')$$

While programs consume input as data, in a weird machine the program is data and the input is the program—an attacker can program the machine by providing input! Consequently, weird machines have interesting properties: an unknown state space, unknown computational power, and an emergent instruction set.

2.2.6 Attacker Models

Suppose we have a method for entering some q_{init} from a set of sane states $\{q_i\}_{i \in I} \subseteq Q_{cpu}^{sane}$. Exploitation is the process of setup (choosing the right q_i), instantiation (entering q_{init}), and programming the weird machine.

We can give an attacker varying capabilities: choosing any bit to flip at any time, restricting bitflips to non-registers, and restricting bitflips to happen at specific program point(s). This abstracts over Rowhammer and buffer overflow attacks, by reducing them to bitflips.

2.2.7 Exploitability

Using our formal definitions, we can prove the security (or lack thereof) of our two variants:

- Variant 1: Not exploitable! We can prove that any bit-flip can be achieved by a finite number of legitimate transitions which stay within Q_{cpu}^{sane}, so the attacker cannot use bit-flips to break security.
- Variant 2: Exploitable. An attacker can set up a data structure so that a bitflip corrupts a pointer, allowing a known value to be treated as a password.

2.3 Weird Machines as Insecure Compilation

One approach for preventing weird machines is to view an exploit as behavior in the target that does not correspond to behavior in the source—that is, insecure compilation. We formally define an exploit and a weird machine as follows:

Definition (Exploit)**.** *An* exploit *of a vulnerable source program* V *is an attacker context* A *from an attack class* $\mathcal{A}$ *if* $Behav\,(C[V]) \neq Behav\,(A[\![V]\!])$ *for every non-oblivious*[1] C*.*

$$Exploit^{\mathcal{A}}(V) \triangleq \left\{ A \in \mathcal{A} \;\middle|\; \begin{array}{l} \forall\, C\,.\, \neg oblivious(C) \;\Rightarrow \\ Behav\,(C[V]) \neq Behav\,(A[\![V]\!]) \end{array} \right\}$$

Definition (Weird Machine)**.** *The* weird machine *of a vulnerable source program* V *for an attack class* $\mathcal{A}$ *is the collection of behaviors arising from exploits of* V*.*

$$WM^{\mathcal{A}}(V) \triangleq \left\{ Behav\,(A[\![V]\!]) \;\middle|\; A \in Exploit^{\mathcal{A}}(V) \right\}$$

This is a generalization of Dullien's approach, using familiar vocabulary of language-based security. In fact, robust hyperproperty preservation exactly describes the absence of weird-machine exploits.

Thus, weird machines provide an example of applying language-based security to understand and reason about vulnerabilities, giving access to more methods for enforcing security.

References

Material in these notes and in Prof. Chong's slides include content from the following publications:

- Patrignani, M., A. Ahmed, and D. Clarke (2019, Feb). Formal approaches to secure compilation: A survey of fully abstract compilation and related work. ACM Comput. Surv. 51(6).
- Abate, C., R. Blanco, D. Garg, C. Hritcu, M. Patrignani, and J. Thibault (2019). Journey beyond full abstraction: Exploring robust property preservation for secure compilation. CSF 2019, pp. 256–271. IEEE.
- Dullien, T. (2020). Weird machines, exploitability, and provable unexploitability. IEEE Transactions on Emerging Topics in Computing 8(2), 391–403.
- Paykin, J., E. Mertens, M. Tullsen, L. Maurer, B. Razet, and S. Moore (2019). Weird machines as insecure compilation. In Workshop on Foundations of Computer Security.