ALE: Awesome Likeable Enclaves

Aaron Bembenek
bembenek@g.harvard.edu

Lily Tsai
lilliantsai@college.harvard.edu

Ezra Zigmond
ezigmond@college.harvard.edu

I. Introduction

Our goal is to formally verify work begun by Gollamudi and Chong, who introduce ImpE, a security-typed calculus that models low-level imperative programs that can directly use enclaves (a sort of private memory region), and provide a translation from a higher-level, enclave-agnostic calculus, ImpS, to ImpE [1]. Our project naturally falls into two parts: proving that every well-typed program in ImpE is secure against a range of low-level attackers, and proving that the translation of a well-typed program written in ImpS results in a well-typed ImpE program that enforces the security policies of the source program.

A. ImpE Security

ImpE is a security-typed calculus that models direct access to enclaves, a hardware mechanism that provides code and memory isolation, to the effect that only code running in an enclave can access the memory associated with that enclave. This isolation guarantee holds even in respect to the operating system kernel [2]. The typing rules of ImpE guarantee that a well-typed ImpE program enforces its security policies in the face of low-level attackers, including those that can arbitrarily modify non-enclave code and, in some cases, even modify enclave code.

The only input to an ImpE program is its initial memory. Thus, all sensitive information is contained in the program's initial memory. A security policy is expressed by assigning each location in the initial memory a security level (*low* or *high*). The program can perform operations on values in the initial memory, and output the results of these computations to a channel with a given security level (*low* or *high*). A program can also declassify a computation result, so that even if the inputs to the computation are from locations at *high* security levels, the result of the computation can be released on a *low* channel.

An attacker with security level *low* can only observe output results on the *low* channel, and if the attacker is at the *high* security level, she can observe events with security level *low* or *high*. From the attacker's observations, she can deduce a set of initial memories upon which the program could execute and produce the same output as what she observed. This set of initial memories is called the attacker's *knowledge*. The smaller this set of possible initial memories, the more precisely the attacker can define the actual contents of memory.

Security is defined as a lower bound on the attacker's knowledge. For example, say the attacker is at level *low* and observes all outputs on the *low* channel. The program executes with initial memory m_0. For a program to be secure, the attacker's knowledge must include any memory m s.t.

1) m contains the same values as m_0 at any locations with a *low* security level.
2) m would have produced the same results as m_0 for any declassified computations.

Our proof that a program is secure at a given security level follows the technique of Pottier and Simonet [3]. We introduce ImpE2, a language that captures the execution of an ImpE program under two different memories. We execute the program in question using ImpE2 with an arbitrary m_0 and a memory m satisfying the properties (1) and (2) above. We show that m must also be in the attacker's knowledge: the attacker's observations at the specified security level when the program executes on m is the same as those when the program executes on m_0. Using this technique, we prove the following three theorems about the security of well-typed ImpE programs:

```
(* well-typed ImpE programs are secure
   against a passive attacker *)
Lemma secure_passive : forall g G G' K' d c,
  well_formed_spec g →
  corresponds G g →
  context_wt G d →
  com_type (LevelP L) Normal G nil nil d c
      G' K' →
  secure_prog L g cstep estep c.
(* well-typed ImpE programs are secure
   against an active attacker
   who cannot modify enclave code *)
Lemma secure_n_chaos : forall g G G' K' d c,
    well_formed_spec g →
    corresponds G g →
    context_wt G d →
    com_type (LevelP L) Normal G nil nil d c
          G' K' →
    secure_prog L g cstep_n_chaos estep c.
(* well-typed ImpE programs are secure
   against an active attacker
   who can modify enclave code *)
Lemma secure_e_chaos : forall g G G' K' d c I,
    well_formed_spec g →
    corresponds G g →
    context_wt G d →
    com_type (LevelP L) Normal G nil nil d c G' K' →
    secure_prog H (g_prime d g I)
         (cstep_e_chaos I) estep c.
```

These three lemmas state the security guarantees of well-typed ImpE programs for each of the three attacker-types we model: purely passive attackers who observe the output trace

of the program, active attackers who can only modify non-enclave code, and attackers who can modify any code. The `well_formed_spec`, `corresponds`, and `context_wt` hypotheses are preconditions to the typing judgment that (intuitively) ensure that the initial security specification, memory specification, and typing context are well-formed and make sense together. The `com_type` hypothesis is the typing judgment for ImpE commands. Finally, the `secure_prog` judgment corresponds to the definition of program security presented earlier: an attacker cannot learn anything more about the initial memories of a program than allowed by the security policy `g`.

B. ImpS to ImpE Translation

ImpS is a security-typed calculus that is very similar to ImpE, except that it is not enclave-aware. A well-typed ImpS program enforces its policies against high-level (observational) attackers, but is not secure against low-level attackers who can modify program code or arbitrarily access the program's memory. Chong and Gollamudi provide a constraint-based translation scheme from ImpS to ImpE that guarantees that the resulting ImpE program is well-typed and enforces the source program's security policies even against these types of low-level attackers.

For instance, consider the following ImpS program, where the memory locations `password` and `guess` have *high* security:

```
status := declassify(*password == *guess);
output status to Low
```

This program is secure against a passive attacker, who just observes whether the guess matches the password (an intentional information leak). However, it is not secure against a low-level attacker that can arbitrarily modify code. This attacker could change the final line to `output *password to Low`, revealing information that the attacker is not supposed to learn. A translation of this program to ImpE would provide security against this attack:

```
enclave(1, status :=
    declassify(*password == *guess));
output status to Low
```

The translation places the memory locations `password` and `guess` in enclave 1, which means that the program would fault if the attacker modified the code to try to access `password` outside of this particular enclave. The fault is captured by the ImpE semantics, which does not allow an enclave memory location to be referenced by code outside of its enclave.

We anticipate that we will model Gollamudi and Chong's translation scheme as a propositional judgment. The alternative would be to try to model their translation scheme as a computation that would take an ImpS program as input and output an ImpE program, but this approach seems more challenging and could potentially require constraint solving. A simplified version of the ultimate theorem we hope to prove is:

```
Theorem translation_sound:
  forall (c: imps_com) (g: imps_env)
         (c': impe_com) (g': impe_env),
    imps_well_typed c g →
    translation c g c' g' →
    impe_well_typed c' g'.
```

II. Schedule

- Apr 17: ImpE2 complete, prove secure_passive.
- Apr 20: Translation code written, lemmas stated and partially proven.
- Apr 24: ImpE2 for nchaos and echaos implemented, working on proof for secure_nchaos and secure_chaos.
- Apr 27: All proofs drafted.
- May 1: Proofs finalized. Start writing paper.
- May 4: Draft of paper complete.
- May 8: DUE

III. Division of Labor

We will do the laborious tasks in a divisive fashion. We see the following as parts that can be implemented in parallel:

- Define ImpE2 for passive attacker and two active attackers
- Security proofs using ImpE2 for all attacker models
- ImpS semantics/syntax, translation specification and proof

We have divided up the semantics/typing judgments/syntax implementation (currently, Aaron is working on the translation and ImpS, and Lily and Ezra are working on defining and proving ImpE2 for the passive attacker model). We believe that collaborating on our proofs (at least at the beginning) will be extremely helpful. Our paper will also be a joint collaborative effort.

IV. Risks and Fallback Plan

We have found that certain modeling decisions made in the Gollamundi/Chong paper are not suitable for a Coq development, or make it more difficult to prove the stated security theorems. Furthermore, their proofs often rely on intuition, and we foresee difficulties expressing these intuitions in Coq.

Our basic goal is to prove security against the passive attacker, as well as the soundness of the translation. Given how this proceeds, we will hopefully prove security against all three attacker models. If we cannot reach our basic goal, our fallback plan is to simplify the ImpE model (for example, allowing an attacker to view the entire execution of the program rather than simply a portion of it).

V. Future Work

Future work includes a computational, rather than propositional, implementation of the ImpS to ImpE translation, and a computational type-checker for both ImpE and ImpS. We would need to prove that these computational versions adhere to our specification and therefore ensure the security properties proven.

Far-future work includes verifying a security-aware compiler pass for CompCert that automatically inserts enclaves

with a given security policy. This is a large step from modeling a simple calculus such as ImpE, and will require significantly more complex modeling.

REFERENCES

[1] A. Gollamudi and S. Chong, "Automatic enforcement of expressive security policies using enclaves," in *Proceedings of the 29th Annual ACM SIGPLAN Conference on Object-Oriented Programming Languages, Systems, Languages, and Applications*. New York, NY, USA: ACM Press, Oct. 2016.

[2] V. Costan and S. Devadas, "Intel sgx explained." *IACR Cryptology ePrint Archive*, vol. 2016, p. 86, 2016.

[3] F. Pottier and V. Simonet, "Information flow inference for ml," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 25, no. 1, pp. 117–158, 2003.