When Good Components Go Bad
Formally Secure Compilation Despite Dynamic Compromise

Cătălin Hrițcu
Inria Paris
Security foundations research is about making this diagram mathematically formal.

1. Security Goal [What are we trying to achieve?]
   - negative definition: What (kind of) attacks are we trying to prevent?
   - positive definition: What security property are we aiming for?

2. Security Enforcement [How can we effectively achieve it?]
   - static: program verification, static analysis, type systems, ...
   - dynamic: reference monitors, hardware mechanisms, crypto, ...
   - trade off security vs. precision, efficiency, compatibility, ...

3. Security Proof [How can we make sure we achieved it?]
TRUST ME. OUR CLOUD SECURITY IS SO GOOD EVEN YOU WON'T BE ABLE TO ACCESS YOUR DATA!
Security proof

• **Marketing snake oil**: trust me, it isss very sssecure
• ...
• **Security experts, metrics, standards**
• **Informal code audit**
• **Security testing**, red teaming, bounty programs
• ...
• **Mathematical proofs** with various levels of rigor
• **Formal, machine-checked proofs**
  – in a proof assistant like Coq, Isabelle, HOL, F*, EasyCrypt, ...
  – about **abstract models** or **concrete implementations**
  – under various **assumptions** and **trusted computing base**
Project Everest is a multiyear collaborative effort focused on building a verified, secure communications stack designed to improve the security of HTTPS, a key internet safeguard. This post—about the proving methodology and verification tools of Project Everest—is the third in a series exploring the groundbreaking work, which is available on GitHub now.
EverCrypt: Verified Crypto Provider

- **Verified C (HACL*)**: ChachaPoly, SHA2+3, Blake2, Curve25519, ...
- **Verified X64 ASM (Vale)**: AES-GCM, Poly1305, Curve25519, ...
- **Very good efficiency**, competitive to libcrypto or libsodium
- **Readable** C and ASM code
- **Deployed in production**
  - Mozilla Firefox (NSS)
  - Microsoft WinQUIC
- **Project Everest**, extending this to:
  - verified TLS implementation
  - verified HTTPS stack
EverCrypt formally

1. Security Goals
   - Memory safety (no buffer overflows, use-after-frees, double-frees, ...)
   - Functional correctness (code implements a simpler math function)
   - Side-channel resistance (secret independent control & mem accesses)
   - Cryptographic security (e.g. auth, int, and conf of AEAD constructions)

2. Security Enforcement
   - static: program verification in F* for safety and correctness
   - side-channel resistance and crypto security involve paper proofs

3. Security Proof
   - milestone: 40.000+ lines of proved correct code, shipping
   - still: big trusted computing base, some interesting proofs on paper
Formally Secure Compartmentalization

When Good Components Go Bad (CCS 2018)
Beyond Good and Evil (CSF 2016)
Core team at Inria Paris

Carmine Abate
Rob Blanco
Florian Groult
Cătălin Hrițcu
Théo Laurent
Jérémy Thibault

Collaborators

Arthur Azevedo de Amorim
CMU (ex Inria)

Boris Eng
Paris 7 (ex Inria)

Ana Nora Evans
U. Virginia (ex Inria)

Guglielmo Fachini
Nozomi (ex Inria)

Yannis Juglaret Quarkslab
(ex Inria)

Benjamin Pierce
UPenn

Marco Stronati
Tezos (ex Inria)

Andrew Tolmach
Portland State
Inherently insecure languages like C

— any **buffer overflow** can be catastrophic

— ~100 different **undefined behaviors**

  in the usual C compiler:

  • use after frees and double frees, invalid type casts, signed integer overflows, ................................

— **root cause**, but very challenging to fix:

  • **efficiency**, precision, scalability, backwards compatibility, deployment
Compartmentalization mitigation

• **Break up security-critical applications** into **mutually distrustful components with clearly specified privileges**

• **Enforce this component abstraction all the way down**
  – separation, static privileges, call-return discipline, types, ...

• **Compartmentalizing compilation chain:**
  – compiler, linker, loader, runtime, system, hardware

• **Base this on efficient enforcement mechanisms:**
  – OS processes (all web browsers) — hardware enclaves (SGX)
  – WebAssembly (modern web browsers) — capability machines
  – software fault isolation (SFI) — tagged architectures
1. Security Goal

[What are we trying to achieve?]

• Hoping for strong security guarantees one can make fully water-tight
  – beyond just "increasing attacker effort"

• Intuitively, if we use compartmentalization ...
  ... a vulnerability in one component does not immediately
    destroy the security of the whole application
  ... since each component is protected from all the others
  ... and each component receives protection as long as
    it has not been compromised (e.g. by a buffer overflow)
Can we formalize this intuition?

What is a compartmentalizing compilation chain supposed to enforce precisely?

Formal definition expressing the end-to-end security guarantees of compartmentalization
Challenge formalizing security of mitigations

• **We want source-level security reasoning principles**
  – easier to **reason about security in the source language** if and application is compartmentalized

• ... **even in the presence of undefined behavior**
  – can't be expressed at all by source language semantics!
  – what does the following program do?

```c
#include <string.h>
int main (int argc, char **argv) {
    char c[12];
    strcpy(c, argv[1]);
    return 0;
}
```
Compartmentalizing compilation should ...

- **Restrict spatial scope** of undefined behavior
  - mutually-distrustful components
    - each component protected from all the others

- **Restrict temporal scope** of undefined behavior
  - dynamic compromise
    - each component gets guarantees as long as it has not encountered undefined behavior
    - i.e. the mere existence of vulnerabilities doesn't necessarily make a component compromised
∃ a sequence of component compromises explaining the finite trace $t$ in the source language, for instance $t=m_1 \cdot m_2 \cdot m_3$ and

(1) $C_0 \xhookrightarrow{} C_1 \xhookrightarrow{} C_2 \xhookrightarrow{} * m_1\cdot\text{Undef}(C_1)$

(2) $\exists A_1. \ C_0 \xhookrightarrow{} A_1 \xhookrightarrow{} C_2 \xhookrightarrow{} * m_1 \cdot m_2 \cdot \text{Undef}(C_2)$

(3) $\exists A_2. \ C_0 \xhookrightarrow{} A_1 \xhookrightarrow{} A_2 \xhookrightarrow{} m_1 \cdot m_2 \cdot m_3$

Finite trace records which component encountered undefined behavior and allows us to rewind execution

Security definition: If $C_0 \xhookrightarrow{} C_1 \xhookrightarrow{} C_2 \xhookrightarrow{} t$ then

∃ a sequence of component compromises explaining the finite trace $t$ in the source language, for instance $t=m_1 \cdot m_2 \cdot m_3$ and
2. Security Enforcement

[How can we effectively enforce this?]

Proof-of-concept
secure compilation chain
Compartmentalized unsafe source

Buffers, procedures, components interacting via strictly enforced interfaces

Compartmentalized abstract machine

Simple RISC abstract machine with build-in compartmentalization

Micro-policy machine

Software fault isolation

Bare-bone machine

Tag-based reference monitor enforcing:
- component separation
- procedure call and return discipline
  (linear capabilities / linear entry points)

Inline reference monitor enforcing:
- component separation
- procedure call and return discipline
  (program rewriting, shadow call stack)

Expectation: other enforcement mechanisms should work as well
Micro-Policies [Oakland’15, ASPLOS ’15,...]

software-defined, hardware-accelerated, tag-based monitoring

```
<table>
<thead>
<tr>
<th>pc</th>
<th>tpc</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>tr0</td>
</tr>
<tr>
<td>r1</td>
<td>tr1</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>mem[0]</th>
<th>tm0</th>
</tr>
</thead>
<tbody>
<tr>
<td>“store r0 r1”</td>
<td>tm1</td>
</tr>
<tr>
<td>mem[2]</td>
<td>tm2</td>
</tr>
<tr>
<td>mem[3]</td>
<td>tm3</td>
</tr>
</tbody>
</table>
```

software monitor’s decision is hardware cached

policy violation stopped! (e.g. out of bounds write)
Compartmentalization micro-policy

Invariant:
At most one return capability per call stack level

Cross-component call only allowed at EntryPoint

Cross-component return only allowed via return capability

Loads and stores to the same component always allowed
3. Security Proof

[How can we make sure we achieved our goal?]

Proof-of-concept *formally secure compilation chain* in Coq
Verified

Compartmentalized unsafe source

Buffers, procedures, components interacting via strictly enforced interfaces

generic proof technique

Compartmentalized abstract machine

Simple RISC abstract machine with build-in compartmentalization

Micro-policy machine

Tag-based reference monitor enforcing:
- component separation
- procedure call and return discipline
(linear capabilities / linear entry points)

Bare-bone machine

Inline reference monitor enforcing:
- component separation
- procedure call and return discipline
(program rewriting, shadow call stack)

Systematically tested (with QuickChick)

21K lines of Coq, mostly proofs

 Verified

https://secure-compilation.github.io
We reduce our proof goal to a variant of:

Robust Safety Preservation

∀source components.
∀(bad/attack) finite trace \( t \).
Simple and scalable proof technique
(for our variant of Robust Safety Preservation)

1. back-translating finite trace prefix to whole source program
2+4. compiler correctness proof (à la CompCert) used as a black-box
3+5. also simulation proofs, but at a single level

Source

\[(m, IC \cup IP) \uparrow = (CS \cup P') \rightsquigarrow^* m\]
\[\quad \Rightarrow m \leq t \lor t < P m \leftarrow 5 \text{ Blame}\]
\[\quad \Rightarrow (CS \cup P) \rightsquigarrow t \land (m \leq t \lor t < m)\]

Target

\[(CT \cup P) \rightsquigarrow^* m \quad \Rightarrow (CS_\downarrow \cup P') \rightsquigarrow^* m \quad \Rightarrow (CS_\downarrow \cup P) \rightsquigarrow^* m\]

1 Back-translation
2 Forward Compiler Correctness
3 Recomposition
4 Backward Compiler Correctness
When Good Components Go Bad

1. **Goal:** formally secure compartmentalization
   - *first definition* supporting mutually distrustful components and dynamic compromise
   - restricting undefined behavior *spatially* and *temporally*

2. **Enforcement:** proof-of-concept secure compilation chain
   - software fault isolation or tag-based reference monitor

3. **Proof:** combining formal proof and property-based testing
   - Generic proof technique that *extends* and *scales well*
Making this more practical ... next steps:

• Scale formally secure compilation chain to C language
  – allow pointer passing (capabilities)
  – eventually support enough of C to measure and lower overhead
  – check whether hardware support (tagged architecture) is faster

• Extend all this to dynamic component creation
  – rewind to when compromised component was created

• ... and dynamic privileges
  – capabilities, dynamic interfaces, history-based access control, ...

• From robust safety to hypersafety (confidentiality) [CSF'19]

• Secure compilation of EverCrypt, miTLS, ...
My dream: secure compilation at scale

C language
+ components
+ memory safety

ASM language
(RISC-V + micro-policies)

language
Going beyond Robust Preservation of Safety

Journey Beyond Full Abstraction (CSF 2019)
Going beyond Robust Preservation of Safety [CSF'19]

relational hyperproperties (trace equivalence)

+ code confidentiality

hyperproperties (noninterference)

+ data confidentiality

trace properties (safety & liveness)

only integrity

No one-size-fits-all security criterion

More secure

current proof technique

More efficient
to enforce

Easier to prove

realistically enforceable?
When Good Components Go Bad

1. **Goal: formally secure compartmentalization**
   - **first definition** supporting mutually distrustful components and dynamic compromise
   - restricting undefined behavior **spatially** and **temporally**

2. **Enforcement: proof-of-concept secure compilation chain**
   - software fault isolation or tag-based reference monitor

3. **Proof: combining formal proof and property-based testing**
   - **Generic proof technique** that **extends** and **scales well**