VeTSS Workshop on Verified Software

Software verification at Arm

Two case studies

Gustavo Petri, Dominic Mulligan
Security Group, Arm Research

© 2019 Arm Limited

Arm Research Security group

Security, privacy, correctness

Mission: *Systematically remove excuses for untrustworthiness*

- Develop new security and robustness technologies, and accelerate adoption by eliminating performance costs and other obstacles
- Reduce what needs to be trusted: minimize the trusted computing base (TCB), and **verify what remains**
- Take a principled and quantitative approach to security through the use of **formal methods**
- Progress the application of privacy-enhancing technologies to information processing systems — demonstrate that business requirements can be addressed while respecting users and their data

© 2019 Arm Limited
arm

Two case studies

Two different uses of formal verification technology:

1. Verification of ultra-low level security-critical firmware

2. Verification in a privacy-preserving compute project

(Original talk title mentioned three case studies, sadly we only have 45 minutes…)

© 2019 Arm Limited
arm

Security-critical firmware verification

Alex Chadwick, Nathan Chong,
Dominic Mulligan, Gareth Stockwell

© 2019 Arm Limited

Background

A security-oriented project under development in Arm

- Limited in what we can talk about…:
- …but example of us using software verification **in production**

Project consists of two components:

- "Traditional architecture": TrustZone hardware
- Firmware component built on top of TrustZone, written in C

Programmer-facing functionality provided by firmware

© 2019 Arm Limited

Firmware challenges

How do we deliver a high-quality product?

- All Arm hardware designs are extensively tested and verified before shipping
- Firmware implementation strategy not an excuse to ship substandard product!

Firmware will be running in a privileged *Exception Level* on processor

- If buggy, could have significant effect on usability and security of system
- Moreover, tempting target for malefactors to try and attack

On the other hand, firmware can't be correct but dog-slow

- Firmware is a product and verification techniques have to fit around that

© 2019 Arm Limited

Commands

Architecture provides a series of programmer-facing **commands**:

- Invoked and take parameters similar to a *sys-call*, with a defined calling convention
- Parameters may be different types of addresses (PA/VA), which point to pages of memory: firmware therefore needs to handle address translation explicitly
- Some pages contain *metadata objects* that are kept isolated, inaccessible to any agent other than the firmware and are used by the firmware as a book-keeping mechanism

Some metadata objects are arranged in tables, some have fields that point to other metadata objects, some control memory permissions, some have a state, and so on

Metadata objects with state have a defined **lifecycle**: transition diagram through which they progress and which must be enforced by the commands

© 2019 Arm Limited

Firmware specification

Firmware's behaviour is well specified by software standards, consists of two parts:

1. ASL "implementation" showing the functionality of the firmware at an abstracted level
2. A set of pre- and postconditions that describe the action of each command using assertions over an abstracted Arm machine state

Like specification and *"meta-specification":* pre-/postconditions have precedence over ASL

ASL implementation very useful, however:

- Exists to check pre-/postconditions can be realized through a command, check internal consistency of architecture, allow integration tests with wider Arm v8-A architecture, as a guide to implementors, as a means of communicating with partners, etc…

© 2019 Arm Limited

Pre-/postcondition specification

Failure: Checked first, and in the order specified, e.g.:

- *"if address A not page aligned, return error code* `Error_Alignment`*"*
- *"if translation of address A fails, return error code* `Error_Translation`*"*
- *"if address A points to structure S, and* `S.type` *is not T, return error code* `Error_Parameter`*"*

Success: Assume no failure modes apply, describe state change effected by command, e.g.:

- *"if address A points to structure S, and S has type T, then change T to type U"*

Implicit *atomicity* pre-/postconditions, as firmware is running in concurrent setting:

- If command fails to obtain exclusive access to resources, abort
- If command aborts for any reason, then no effect on system is observed
- Any synchronization mechanisms obtained (e.g. locks) are eventually released

Note, particular synchronization strategy used is not *architecturally* prescribed

© 2019 Arm Limited

Specification internal consistency

For Arm, **specification is as-much the product as implementation!** First step is checking specification is not broken and can be implemented…

- All pre-/postconditions for commands are written in *JSON*, used as common source for:
  1. SystemVerilog assertions (SVA henceforth)
  2. CBMC assertions
  3. English-language stylized rendering of pre-/postconditions that will appear in reference manual
- From ASL prototype implementation of each command, we can also extract Verilog using *ArchEx*, an Arm-internal tool for working with ASL
- We then feed both the SVA and the autogenerated Verilog into a commercial model checking tool

(Finds *lots* of bugs: architects make changes to specification, repeat over many months…)

© 2019 Arm Limited
arm

Model Checker verification testbenches

Various testbenches (due to Alex Chadwick):

- **Lint**: checks all inline assertions in the ASL implementation are always reachable, and can never be invalidated
- **Pre-post**: check ASL implementation against declarative description of pre-/postconditions for each command
- **Lifecycle**: checks lifecycle transitions of metadata object match declarative description of possible state transitions
- **Invariant**: checks higher-level invariants of machine state are never invalidated

All run paths of *N* arbitrary commands out of a reset state, with a "*havocking*" step which perturbs register and memory contents between each invocation

© 2019 Arm Limited

Verifying the implementation

Once specification is stable, need to verify C firmware implementation

We use *CBMC*, the C bounded model checker, for this task:

- It's generally robust, and is good at parsing C code that's written by real engineers
- *Counterexample-guided debugging* is very useful
- In our context, *model checking* is familiar and engineers quick to understand CBMC
- C programmers also quickly pick CBMC up with minimal training and use existing C expertise to understand (and write!) the properties being verified

© 2019 Arm Limited
arm

Toward a command testbench

Suppose command `Foo(A)` has postcondition (with A a VA):

if address A points to structure S, and S has type T, then change T to type U

Unpacking this a little:

- Testbench assumes there's a structure S floating around in physical memory somewhere, that command will modify. Call this S the command **footprint**
- A is a VA and will point to structure S only after address translation into PA, P. We therefore need to model Arm address translation…

Address translation is complicated: having to model it is not good… Can we avoid that?

© 2019 Arm Limited
arm

Not modelling address translation

First, the command footprint is declared in the testbench preamble:

```
struct S s;
```

and (simplifying) this is going to have various addresses associated with it:

```
struct footprint {
  paddr_t pa; vaddr_t va; bool is_locked;
};
```

Initially set `pa`, `va`, and `is_locked` fields to be **unconstrained** values:

```
footprint.pa = ★; footprint.va = ★; footprint.is_locked = ★;
```

Here, ★ is a rendering of CBMC’s **non-deterministic assignment**…

© 2019 Arm Limited

arm

Modelling all address translations

Firmware uses a (potentially failing) function with this prototype to translate addresses:

```
paddr_t translate_va_to_pa(vaddr_t va)
```

Our strategy is to replace the implementation of this function (that does *Arm v8A AT*) with our own address translation implementation

Rather than model Arm AT, however, we model **all** such potential address translations and show firmware implements correct pre-/postcondition functionality in all cases

Stronger property, and also easier to verify

© 2019 Arm Limited
arm

Using uninterpreted functions

We can model **all** such functions using two **uninterpreted functions**:

```
bool UF_translate_va_to_pa_status(vaddr_t addr)
paddr_t UF_translate_va_to_pa(vaddr_t addr)
```

First signals that address translation at VA `addr` failed, the second computes the AT

Don’t know return value of functions, but will always be same whenever same input given

CBMC allows us to attach explicit **assumptions** to these functions, constraining behaviour

© 2019 Arm Limited
arm

Assuming address translation into life

We now *"assume a translation into existence"*, by having this in our testbench preamble:

```
if(UF_translate_va_to_pa_status(footprint.va)) {
      assume(footprint.pa == UF_translate_va_to_pa(footprint.va))
}
```

Footprint's PA and VA's linked by AT *if AT succeeds*. We implement the firmware AT API:

```
paddr_t translate_va_to_pa(vaddr_t va) {
  return UF_translate_va_to_pa(va);
}
```


© 2019 Arm Limited
arm

Mocking physical memory

Firmware abstracts physical memory with read/write API for each metadata object type, S:

```
void readS(paddr_t addr, struct S* const obj)
void writeS(paddr_t addr, const struct S* const obj)
```

Raw pointers into physical memory are never dereferenced other than via this API

Strategy: implement this API in order to observe what physical memory reads/writes a command is doing, write *CBMC* assertions over the contents of physical memory

Bonus: we can use *CBMC's* static analysis to guarantee that the firmware is *only* writing to memory via these functions, and not arbitrarily dereferencing raw pointers

© 2019 Arm Limited
arm

Modelling physical memory

Can pin down a command's access to memory tightly:

```
void readS(paddr_t addr, struct S* const obj) {
  if(addr == footprint.pa) { *obj = s; }
  else { UNREACHABLE; }
}
```

CBMC checks UNREACHABLE really is for a command: if not the command is doing something weird. This can be really pinned down:

- If object is only read, not written, the write function is UNREACHABLE for all inputs
- If a command only needs to access field `f` in a struct, then we can check this, too

© 2019 Arm Limited
arm

Property checking

What we actually check

For any number of metadata objects, for all translations, for the full address space:

- Commands implement the declarative pre-/postconditions specification detailed in the common JSON specification,
- If a command takes locks, then all locks are released,
- If a command fails then it rolls back all state changes,
- All pre-/postconditions are reachable and there is no over-constraining (using CBMC's reachability checking), moreover testbench assumptions are not contradictory (by asking CBMC to prove *false* within the testbench context)
- Commands terminate on all inputs (using CBMC's unwinding assertion checking, loops are bounded, no recursion)
- Firmware code is free of some undefined behaviours (using CBMC static analysis)

© 2019 Arm Limited

Are we really checking the right thing?

Testbenches require additional code: are we really checking the right thing, then?

Key idea: deliberately inject bugs into firmware codebase to test a testbench

In our case, interested in whether potential bugs trigger a testbench assert failure:

- Mutants that fail indicate the asserts are capturing the right functionality
- Mutants that do not fail indicate our properties may not be strong enough

Some simple mutations are enough to capture some interesting bugs:

- Flipping conditional checks, removing `continue` statements, decrementing rather than incrementing, etc.

© 2019 Arm Limited

Bug caught using mutation testing

Assumptions on address translation too strong. Previously:

```
assume(footprint.pa == UF_translate_va_to_pa(footprint.va))
```

Now:

```
if(UF_translate_va_to_pa_status(footprint.va)) { assume(…) }
```

Ensures that if firmware ignores the translation status and attempts to use the output PA then the testbench will fail when we try to manipulate the object at the bogus PA

© 2019 Arm Limited
arm

Deployment

All of this machinery is in active use with our product group:

- *CBMC* is integrated into *Jenkins CI*: a *CBMC* smoketest consisting of a static analysis run and reachability checking is run on every pull request, with a full verification run nightly
- One-day course on *CBMC* delivered to engineering teams by Nathan, Dominic
- *JSON* specification file and testbenches for new commands managed by product group engineers
- Caught some very interesting bugs in both implementation and specification. Interestingly: code had been code-reviewed by two/three engineers…
- Combination of *Hardware Model Checker* and *CBMC* seemed to work very well: worked on different parts of product, worked in slightly different ways, caught different things
- Software verification, and *CBMC*, proved its worth…

© 2019 Arm Limited

arm

Veracruz

Privacy-preserving compute

Derek Miller, Dominic Mulligan, Hugo Vincent, Shale Xiong

© 2019 Arm Limited

Veracruz in a slide

Experimental privacy-preserving compute infrastructure in Arm Research

Most general setting:

- **Secret data** can be fed into,
- **secret programs**, which are offloaded to,
- third parties who **host** the computation, and produce
- **secret results,** retrievable by an agent specified in a **global policy**

In this setting, everybody is *mutually distrusting* and has individual concerns:

- Data and program owners want to retain their secrets,
- Host does not want machine to be damaged by programs they cannot audit or monitor

Uses *"Enclaves"* (based on a hardware-provided TEE) and sandboxing VM to achieve this…

© 2019 Arm Limited

Secure Enclaves, a brief overview

A kind of "inverse sandbox”——enforces **confidentiality**, **integrity** of contents

- Arm TrustZone was first widely-deployed hardware support for *Trusted Execution Environments*
- Also *Intel SGX*, *Sanctum* and *Keystone* (RISC-V), *AEGIS*, *Komodo*, and others…
- Arm looking into standardized enclave abstraction on *AArch64*

Common adversary model, package of features:

Wider execution environment considered hostile, buggy, compromised

- Including operating system, hypervisor, and other systems software

Hardware protected regions of memory

- Protect programs from unauthorized accesses/modifications by privileged code, device *DMA* accesses
- Some enclave schemes offer stronger guarantees (e.g. memory encryption and integrity protection)

Cryptographic **measurement** and **attestation protocol**

- Establishes (remote) enclave correctly initialized with expected program, configuration parameters

© 2019 Arm Limited

arm

Remote attestation, conceptually

Attestation Service

Challenging Party

Verifying Party

Prove that program P with hash **H** is installed in a genuine Enclave with settings **S** on your machine

Instructs hardware to compute **attestation report** containing hash of code loaded into Enclave, details of settings, signed by manufacturer provisioned keys

Tell me if this report:

- Is from genuine hardware you manufactured?
- States that code with hash **H** is installed in the Enclave?
- States the Enclave was configured with settings, **S**?

Validates or **rejects** the report as genuine, or not corresponding to **H**, and **S**

© 2019 Arm Limited
arm

Building little islands of trust

Aim to establish "*island of trust*" on an untrusted remote machine:

1. Upload program to remote machine
2. Remote machine configures and initializes Secure Enclave
3. Demand proof that Enclave contains expected contents, parameters acceptable
4. Validate proof

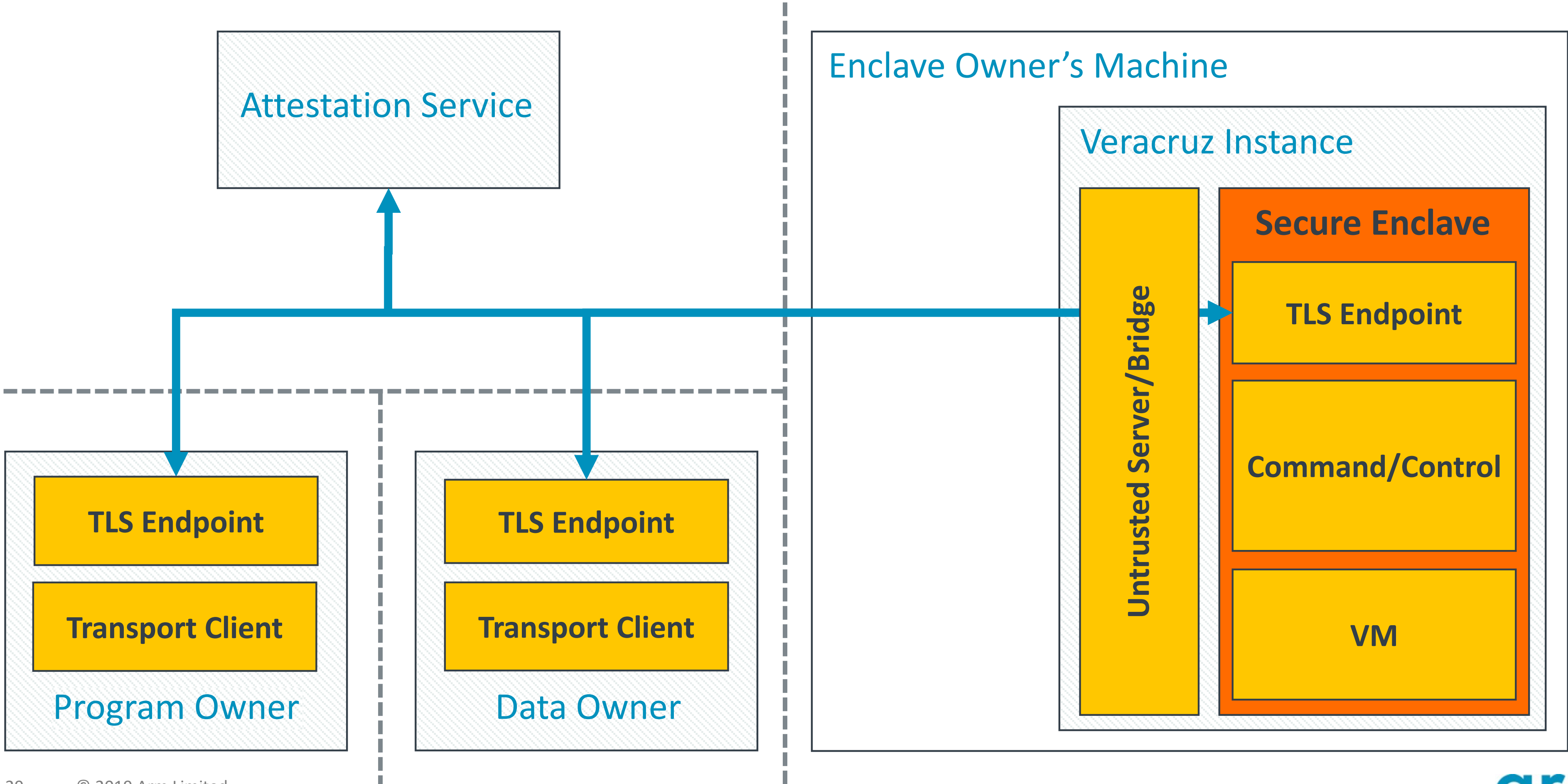
Now safe* to communicate secrets to Enclave:

- Attestation protocol establishes Enclave contents, configuration as expected
- Guarantees of implementation ensure **privacy** and **integrity** of computation

* Providing you trust the hardware and attestation process…

© 2019 Arm Limited
arm

System design



© 2019 Arm Limited
arm

What does this have to do with verification?

Secure Enclaves are not magic…

- **Privacy** and **integrity** guarantees provided are conditional on software being "*well-written*" (for some specific value of "*well-written*" detailed in a **threat model**)
- In particular, memory errors, synchronization errors, side-channels, have all been used to attack Enclaved systems

When building a system using Enclaves there's therefore a **strong** incentive to have:

- As little code as possible running in the Enclave,
- What code there is should be statically analyzed for memory errors and other issues, or written in "memory-safe" languages like Rust,
- What code there is should be simple, and easily auditable by hand…

© 2019 Arm Limited

…moreover

Consider Veracruz's threat model: everybody is assumed hostile to everybody else, only point of trust in system being the hardware and attestation process

Means the program owner, collaborating with the host, is trying to steal secret data provisioned into the Enclave. How could they do this?

Assuming no overt way of doing this is provided by the VM (!), the two have to exploit some unforeseen aspect of the VM's design to "*escape the sandbox*" to achieve this

© 2019 Arm Limited
arm

Building trust through verification

Strategy: use verification and static analysis of key components in the Enclave as:

- A means of *eliminating memory errors and other trivial language-level bugs*: so that Veracruz code cannot be easily undermined using e.g. **ROP-attacks** by a hostile host,
- *Building trust*: here's what we think the code should be doing, and here's proof that it's doing it——you can check too, if you don't trust us,
- Ensuring *auditability* of Veracruz's code: code that's easy to verify also tends to be easy to manually audit: verification tools struggle on code-bases that are too large, too messy, too clever. **So do humans…**

Code that is not verified should be written in a memory-safe language, like **Rust**, and if something can feasibly be outside the Enclave, then it should be

© 2019 Arm Limited
arm

Veracruz implementation

Sandbox VM prototype implemented in C, *Zocalo* bytecode opcodes taken from WASM:

- VM (~7,000 LOC), uses a simple *fetch-decode-execute* cycle, easy to understand, audit, and retarget compilers to
- Sandboxes program and insulates host machine from running program: no ability to e.g. perform I/O or other side-effects (other than sample a random source)
- Veracruz Executable (VX) files loaded by machine, simplified version of ELF
- *Future*: have VM enforce dynamic policies on running program, e.g. resource limits, trace properties of program, explore impact of/defences against **Rowhammer**, etc.

© 2019 Arm Limited

CBMC verification

Mostly due to Shale Xiong

Again, we use CBMC to verify properties of VM:

- **Static analysis**: no memory issues, no undefined behaviours,
- **Verification**: VM fetch-decode behaves as expected, VM (non-floating point) instruction semantics behaves as expected, various reachability properties of code, various invariants are always maintained (e.g. program counter always points inside code memory)

About 12,000 lines of testbench, takes around 24 hours to verify to completion

Also in the process of getting *TIS-interpreter* set up on the VM codebase, exploring fuzzing

 © 2019 Arm Limited

Future work

A few strands of ongoing/potential future work:

1. **Ongoing**: generalizing to N-sources of secret data to support multi-party computations
   - Opens up some interesting applications of Veracruz, e.g. private map-reduce, private federated machine learning, etc.
2. **Speed**: bytecode is interpreted at the moment, and incurs a runtime overhead
   - For small computations, this is acceptable but can try and JIT the bytecode in the Enclave
   - Challenge: do this in a high-assurance way…
3. **Further reductions in TCB**
   - Only the VM is verified at the moment. TLS library is not: could try to reuse existing verified TLS implementations in the Enclave
4. **Applications**: what can we do with Veracruz?
   - Lots of potential in building more complex systems on top of Veracruz, using it as a component…

© 2019 Arm Limited

arm

arm

Conclusions

© 2019 Arm Limited

Conclusions

Arm are applying more verification techniques, both in products and R&D projects:

- Hardware verification techniques are already extensively used
- Software verification techniques are starting to be deployed, too

Some of these have originated in the Security Group in Arm Research, but:

- Product groups are adopting this capability,
- …and independently developing their own capability, too

Projects described in this talk are just a few of our verification-related projects, e.g.…

© 2019 Arm Limited

Secure-M: the missing case-study

Formal validation of the Arm v8-M microcontroller architecture

- Project led by **Alastair Reid** (now at Google)

Two sub-projects built around custom model-checking techniques using SMT:

- Information flow tracking to detect confidentiality and integrity violations: requires dynamic labelling and explicit declassification and endorsement policies to adequately capture information flow in a microprocessor without false positives
- Internal consistency checks of the v8-M architecture, ensuring inline assertions are not invalidated, and translating English-language rules into formal assertions
- See "*Who guards the guards? Formal validation of the ARM v8-M architecture specification*", **OOPSLA** 2017, by Alastair for more information on rule-checking

© 2019 Arm Limited

Collaboration and internship opportunities

© 2019 Arm Limited

arm

arm

Thank You

Danke

Merci

谢谢

ありがとう

Gracias

Kiitos

감사합니다

धन्यवाद

شكرًا

תודה

© 2019 Arm Limited