A High-Level Overview of SABLE, a Modern Secure Loader

Scott Constable, Robert Sutton, and Stephen Chapin*

Syracuse University

Abstract

The Syracuse Assured Boot Loader Executive (SABLE) aspires to be the world's first formally verified, and independently formally verifiable secure loader. Several modern loaders such as OSLO[5] are merely trusted loaders; they measure code and execute it unconditionally. A secure loader differs from a trusted loader in that it executes code only if measurements of the code match known-good values[12]. In addition, the Syracuse University team plans to apply a rigorous formal verification technique first demonstrated by NICTA in their verification of the seL4 microkernel[7]. This paper summarizes our overall design philosophy from a high level. We present claims about security as well as a wide variety of features, without delving into a great deal of detail. This is on purpose, as we intend to cover the gaps in later papers.

1 Introduction and Background

1.1 Defining a Secure Loader

No computing framework should be considered secure if it lacks a secure loader. Though many such loaders already exist in the open-source world, none is sufficient to meet the increasingly stringent demands of modern cyber security. Trusted Grub, while extremely versatile, is encumbered by a massive Trusted Computing Base (TCB). Trusted loaders such as OSLO rectify this issue with a Dynamic Root of Trust for Measurement (DRTM) model trust chain[5]. Yet they in turn fall short in terms of security. For example, OSLO will unconditionally execute a hypervisor or OS, even if it has been compromised by an attacker.

A secure loader must, by definition, prevent the execution of untrusted code[12]. On the x86 architecture the optimal tool to accomplish this task is the Trusted Platform Module (TPM). By means of cryptographic hashing, the TPM can "measure" code prior to its execution. Additionally the TPM may "seal" data to a particular system state, as defined by hash chains aggregated in secure storage[12]. By fusing these two paradigms, we may formally declare SABLE to meet the criteria for a secure loader.

1.2 Dynamic Root of Trust

The Trusted Computing Group (TCG) somewhat idiomatically uses the term "measurement" to describe a cryptographic hash operation[1]. Measurements can, among other purposes, be used to verify the integrity of code/data or to attest to a particular system configuration. TPM chips contain several Platform Configuration Registers (PCRs) which store measurements. Measurements, however, are not written directly into a PCR. Rather, they are *extended* into a PCR in the following way:

$$\text{PCR} \leftarrow H(\text{PCR} || H(data))$$

where H is a cryptographic hash function, and $||$ is the concatenation operator. That is, the current value in the PCR is replaced by the hash of the old value of the PCR concatenated with the hash of the new data.

This extension scheme is extremely useful, as it allows one to chain hashes together noncommutatively. Hence a single PCR can hold a hash digest of arbitrary length. The application of this technique to secure boot is obvious. By measuring boot components and extending the resulting hashes into one or more PCRs according to their order of execution, we have a complete record of exactly what code was run, and in what order. A bad PCR value could indicate that the system has in some way been compromised during the boot process.

Yet such an elegant scheme rarely comes without flaws. Relying on a hash digest of the entire boot process is actually rife with problems. Low-level software such as the BIOS is generally susceptible to a wide range of attacks, and as such is not infrequently compromised. It may also be that the binaries of these components change over time, and as a result their hash must change as well.

*This research was supported in part by a subcontract from Critical Technologies Inc., under United States Air Force Research Laboratory (USAFRL)- Information Directorate prime contract #FA8750_13_C_0152, based upon US Department of Defense (DoD) Small Business Innovation Research (SBIR) topic #AF121-051, "Remote Attestation and Distributed Trust in Networks (RADTiN)".

[1]TPM chips use SHA1 as of the TPM v1.2 specification[15]

This could make it difficult to keep track of which measurements are "good" and which are "bad."

A relevant concern to security experts is the size of the TCB when accounting for the entire boot process. If we measure all code which is executed, and we rely on all of those measurements to determine our level of trust in the system, then the entire system is the TCB. This is not optimal. To mitigate, the TCG has branched into two separate philosophies for constructing a TCB:

> The starting point of measurement is called the root of trust for measurement. A static root of trust for measurement begins measuring from a well-known starting state such as a power on self-test. A dynamic root of trust for measurement transitions from an un-trusted state to one that is trusted[13].

The suboptimal scenario we described above corresponds to the static root of trust for measurement (SRTM). To minimize the TCB, we would prefer to delay the root of trust as far into the boot process as possible. This is the goal of the dynamic root of trust for measurement (DRTM), also called a late launch.

Both AMD and Intel have implemented DRTM support in their newer x86 architectures. For now, the SABLE team is focusing on AMD's architecture, Secure Virtual Machine (SVM)[2]. However the two are conceptually very similar. SVM provides a special secure machine instruction, `skinit`, which triggers the late launch. The analogous instruction on Intel's Trusted Execution Technology (TXT) architecture is `senter`.

In brief, `skinit` takes as an argument a Secure Loader (SL), and extends its hash into a PCR. This establishes the root of trust for measurement. The SL is then executed unconditionally in solitary confinement: during execution, all but one CPU core is disabled, global interrupts are disabled, hardware debugging is disabled, etc[1]. The lone CPU core in execution during an `skinit` is often referred to as the bootstrap processor (BSP).

One can think of `skinit` as the "big bang." Whatever came pre-`skinit` has no bearing on the execution of the system post-`skinit`. As a result our TCB has been substantially shrunk.

Each v1.2 TPM chip has at minimum 24 PCRs[15]. PCRs 0-15 are generally used for the SRTM model. When the system is reset, these PCRs are all reset to 0. On system boot, PCRs 0-5 are extended implicitely with the BIOS code, BIOS configuration, ROM BIOS, ROM configuration, Initial Program Loader (IPL) code, and IPL code configuration, respectively[2]. The remaining SRTM PCRs may be extended explicitely by the IPL or code which runs after the IPL.

PCRs 17-22 are reserved for the DRTM model. On system reset they are reset to -1. The only way to reset these PCRs to 0 is to fire the `skinit` instruction. One cannot force a hash collision $-1 \rightarrow 0$ because the TPM will reject any request to extend PCRs 17-22 before `skinit` has been fired. PCR17 is of particular interest because it can *only* be extended with the hash of the SL. Since this is purely a hardware procedure, any value in PCR17 which is not -1 must have come directly from the CPU[2]. PCRs 16 and 23 are special-purpose registers, which do not factor into our design.

To prevent any confusion, we should note the distinction between SABLE and the SL. The SL is a sequence of security-critical code which is executed in a hardware protected environment. It is defined within a fixed-size memory region called the Secure Loader Block (SLB). SABLE's job is to prepare the system for `skinit`, then fire `skinit` with the SL as an argument. When the SL has finished, SABLE reinitializes the Application Processors (APs, the cores which are not the BSP) and jumps to the next procedure in the boot chain. So one can either think of the SL as a part of SABLE, or as a separate entity which SABLE calls to perform a sequence of security-critical tasks. For a more expansive discussion on the SL, see Section 3.2.

1.3 Protected Storage

In addition to measurement and integrity reporting capabilities, the TPM offers protected storage capabilities. In particular, a small segment of protected non-volatile memory is reserved for two notable keys. The Endorsement Key (EK) is a key pair which is pre-installed by the manufacturer; it is unique to each TPM. When one takes ownership of the TPM, the Storage Root Key (SRK) is generated by the TPM. Neither of these keys may ever leave the TPM[14].

The EK is generally used for attestation purposes, for instance in signing a report containing the values currently residing in a set of PCRs. Of greater interest to the SABLE project is the role of the SRK. The two primary TPM mechanisms for providing secure storage, binding and sealing, both employ the SRK[14]. Performing a `bind()` operation on some data simply encrypts that data with the TPM's private SRK. Since the SRK, like the EK, is effectively unique to each TPM, in essence this *binds* the data to a particular TPM, and thus a particular machine.

The functionality of the `bind()` operation is further enhanced by the `seal()` command. When data is *sealed*, it is both bound to the platform by the SRK, and bound to a particular system state, as given by a set of PCR values. The `seal()` command takes as parameters the data to be encrypted, and a set of PCR indices[3][12]. The command outputs the cyphertext encrypted by the SRK, as well as an intergrity-protected list of PCR indices and their corresponding values, as specified during the `seal()` call. Together, the cyphertext and PCR list comprise an

[2]AMD recently rebranded SVM as "AMD-V." We still refer to it as SVM to maintain consistency with cited documentation.

[3]The caller will specify whether (a) the data will be sealed to the values currently residing in the given PCRs, or (b) the data will be sealed to some hypothetical PCR values, which the user must also provide as part of the call.

encrypted blob which may be passed into the `unseal()` command. The unsealing will succeed if the PCR values given in the integrity-protected list match the actual PCR values at the time `unseal()` is called[12]. If all of the values match, then the TPM uses its private SRK to decrypt the data.

1.4 Formal Verification

Despite the assurances provided by secure hardware and exhaustive testing, these alone may not be sufficient for deployment in a security-critical setting. A formalized, mathematical proof of a program's correctness can be considered the strongest possible argument for that program's immunity to common vulnerabilities. Until just recently, formal methods proved infeasible in the verification of large and complex programs such as operating systems. Gerwin Klein has compiled an extensive overview of the shortfalls and successes of verification efforts in this area[6].

Yet in recent years, the formal verification of large-scale projects has become not only feasible, but also both time and cost efficient. That is, when one considers the ultimate benefits of a verified final product. The seL4 team at NICTA prototyped, verified, and implemented a high-performance microkernel in an estimated 20 person years[7]. Their final product has been proven void of a number of common bugs and vulnerabilities, such as buffer overflow attacks. Perhaps most impressively, the seL4 microkernel's performance is demonstrably on par with that of several other high-performance microkernels in the L4 family[7]. Following in the footsteps of the seL4 team, the SABLE team will use many of their development techniques to prototype, verify, and implement a secure loader.

All of our formal verification will be done in the theorem prover Isabelle/HOL. Isabelle's efficacy in verifying software has already been demonstrated by the seL4 team[7]. Yet the decision to use a theorem prover complicates the development process. In particular, we must decide whether to start with the prover, or with the implementation. Starting by building a model of the loader in the theorem prover would be impractical, as one cannot interface Isabelle with external emulators. Hence testing the model would be impossible. Starting with the C implementation would also not suffice. Although we can parse C code directly into Isabelle[16], the resulting theorem prover code may be difficult to reason about.

So instead of starting with either the abstract or the concrete, we take a middle-of-the-road approach and build a prototype of the loader in the purely functional language Haskell. Though it may at first seem counterintuitive to model imperative code with a functional language, their relationship is rooted in the state monad. We reserve a lengthy discussion on this topic for a later paper.

Haskell has all of the features we require. It seamlessly interfaces with hardware emulators, is type-safe, and it has a translator, Haskabelle, which converts from Haskell code to Isabelle/HOL code. So we can build and test a model in Haskell, convert it into the theorem prover, prove assertions about the loader, and through refinement[16] extrapolate these assertions to a C implementation. A more detailed overview of this process is given in Section 3.4.

2 Problem Statement

2.1 Threat Model

We divide attacks on SABLE into three categories. The first category of attacks are those which occur before `skinit` is fired. These could involve, for instance, running SABLE from a malicious general-purpose loader, or avoiding SABLE altogether and loading an imposter hypervisor or OS. In general, attacks in this category are allowed to attempt anything at the software level, but nothing at the hardware level.

The second category of attacks occur during the execution of the SL. Such attacks may include anything at the software level. In addition, simple hardware attacks such as using a hardware debugger or power cycling the machine are allowed. But an attacker may not perform sophisticated lab attacks.

The third category comprises attacks which follow execution of the SL. We cannot make any guarantees about these attacks. The reason is that, although SABLE can verify the integrity of the hypervisor when it is launched, SABLE cannot verify that this integrity will be maintained during execution. Once SABLE has terminated successfully, this responsibility will be passed on to the hypervisor. Therefore the hypervisor must aptly maintain its own integrity, along with that of any trusted VMs.

2.2 Goals

A modern secure loader must not simply overcome the limitations of its predecessors. It must transcend them. When SABLE is complete, it will

- be ultra lightweight, perhaps less than 1,000 LoC (Section 4.1).
- be secure, in the sense that the secure framework (e.g. hypervisor/OS) is cryptographically ensured to launch only in the case that it has not been tampered with. The same will hold for sensitive data sealed by the user.
- be both installed and updated at the OS level, with minimal effort by the client (Section 4.3).
- be void of any tedious computations on account of the TPM (Section 4.2).
- be highly resistant to Denial of Service (DOS) attacks (Section 4.5).

- be immune to replay attacks performed on satellite machines, and highly resistant to replay attacks on the local machine (Section 4.6).
- be both formally verified, and independently formally verifiable.
- "run the manual," meaning that the source code *is* the documentation, and vice-versa (Section 4.7).
- have *all* of the security-critical actions be executed within the SL. This implies that the results of computations such as hashing and unsealing will be invisible to all except the adversary who performs a complex physical lab attack on the client's TPM.
- be able to support nested DRTM boots (Section 4.4).
- be able to support Flicker[10] (Section 4.4).

3 The High-Level Specfication

Though SABLE could conceivably load any hypervisor or OS, for the purposes of this project our aim is to load the Genode hypervisor. Loading Genode ourselves from disk, however, is impractical and unnecessary. This would bloat the code base, hinder the formal verification effort, and force us to use more assumptions about hardware/driver correctness. Instead we will employ a mainstream boot loader, and configure it to load our loader, the Genode kernel, the Genode modules, and any necessary configuration files into memory. This will not compromise the integrity of the system, or the security of our loader. This is because the late launch philosophy assumes that everything prior to execution of the DRTM instruction may be untrusted. So since we need not trust the initial boot loader, we will use an extremely robust and flexible loader: Grub 2.

3.1 Installation and Updating

The finer details of this process will be decided upon later. We would very much like to run this process within a Flicker module for maximum security and isolation. In case the reader is not familiar with Flicker, he/she is encouraged to consult [9] and then [10].

1. User provides the new (or updated) loader and Genode to the install routine. The parts we really care about here are the Secure Loader (SL), the Genode kernel (G), the Genode modules (M), and the configuration files (C).
2. Perform $K \leftarrow$ TPM_GetRandom to randomly generate a key.
3. Perform $R \leftarrow$ TPM_GetRandom. R is a value used to prevent local replay attacks.
4. Using a software-based symmetric encryption protocol (ie. AES), encrypt G, M, and C with K to obtain ciphertexts c_G, c_M, and c_C.
5. Compute $h_{c_G} \leftarrow H(c_G)$, $h_{c_M} \leftarrow H(c_M)$, and $h_{c_C} \leftarrow H(c_C)$, where H is the SHA-1 algorithm being run on the CPU (BSP if we elect to use Flicker).
6. Move the ciphertexts to disk.
7. Again using software, compute
$$\begin{aligned}\text{BootRecord} \leftarrow & H(H(H(0||H(0||SL)) \\ & ||H(H(H(0||h_{c_G})||h_{c_M})||h_{c_C})) \\ & ||R).\end{aligned}$$
8. Using the TPM, compute
$$c_K \leftarrow \text{TPM\_Seal}(K, \text{BootRecord})$$
and
$$c_R \leftarrow \text{TPM\_Seal}(R, H(0||SL)).$$
Further request that c_R may only be unsealed by PCR17.
9. Call `TPM_NV_DefineSpace`. This command allows us to specify a region of TPM non-volatile storage which may only be accessed under certain conditions. In particular, we will allow an area the width of c_K to be read and written only if PCR15 = BootRecord[4]. Also allow an area the width of c_R to be written only if PCR15 = BootRecord. We additionally specify that both of these memory regions may only be read once per boot; otherwise an attacker could retrieve these values after SABLE has finished executing.
10. Write c_K and c_R to the TPM's non-volatile storage with `TPM_NV_WriteValue`.
11. Prompt the user to enter a catch phrase (see Section 3.3). Seal this with PCR15, and push it to either the TPM or the hard drive.
12. In the same way that we sealed and stored the symmetric key, we could seal and store other vital information, or at least specify a region in which to store such information. This could be anything from hash chains to monotonic counters.

3.2 The SL

Given below is a step-by-step overview of the actions taken by the SL. Note that all code that is executed in such an environment will have the following characteristics: a) only the BSP is running, b) global interrupts are disabled, and c) hardware debugging is disabled.

1. PCR17 is implicitely extended with the hash of the SL.
2. Check that the values in PCRs 6-15 are all 0. We will use some subset of these PCRs to support nested boot and Flicker (see Section 4.4).

[4]The reason for our choice of PCR15 is outlined in Section 4.4.

3. Retrieve the encrypted replay prevention value c_R from the TPM by executing `TPM_NV_ReadValue`.

4. Obtain $R \leftarrow \text{TPM\_Unseal}(c_R, \text{PCR17})$.

5. Examine the boot headers (provided by GRUB) and extract the memory locations where Genode, its associated modules, and associated configuration are loaded. These are ciphertexts c_G, c_M, and c_C, respectively.

6. Compute $h_{c_G} \leftarrow H(c_G)$, $h_{c_M} \leftarrow H(c_M)$, and $h_{c_C} \leftarrow H(c_C)$. This is done on the BSP, not on the TPM.

7. Extend PCR19 with the hashed values in their respective order. Thus the final value of PCR19 will be

$$\text{PCR19} \leftarrow H(H(H(0||h_{c_G})||h_{c_M})||h_{c_C}).$$

8. Extend PCR15 with the value in PCR17, and then the value in PCR19. So PCR15 will contain

$$\begin{aligned}\text{PCR15} \leftarrow & H(H(H(0||H(0||SL)) \\ & ||H(H(H(0||h_{c_G})||h_{c_M})||h_{c_C}))||R),\end{aligned}$$

   where SL is the binary of the Secure Loader. Notice that this value is precisely that of BootRecord, from Step 7 of the prior section. Hence this register has a non-resettable record of all security critical software that is executed from the instant the `skinit` instruction is fired, to when Genode begins launching VMs.

9. Retrieve the ciphertext symmetric key, c_K, from the TPM by executing `TPM_NV_ReadValue`.

10. Run $K \leftarrow \text{TPM\_Unseal}(c_K, \text{PCR15})$

11. Again using the same symmetric software algorithm, Decrypt c_G, c_M, and c_C with K.

12. Henceforth we will not need the hash value currently residing in PCR19. So we generate a random 20-byte string, and extend this value into PCR19.

At this point, we exit the SL. If anything went awry, this would indicate that some part of the system has been compromised. All errors will be caught at this point, and thrown to the user. The user will be informed of specifically what went wrong. He will then be allowed to reboot and either load into an untrusted OS or else attempt a fix. If nothing went awry, then we can perform any necessary cleanup, and jump to the hypervisor. The period between SL completion and the initial execution of the hypervisor may also be an opportune time to incorporate some form of multi-factor authentication.

3.3 Catch Phrase: User-Based Attestation

Suppose that some untrusted software launches prior to SABLE and installs and runs a version of Genode. This imposter Genode seems to the user exactly like the trusted Genode, except that it actually has malicious intentions. How can the user know that he is in an unsafe environment?

There is already at least one red flag. If the user had previously sealed some sensitive data while in the trusted Genode environment, then he would not be able to unseal that data in the malicious environment. This is because the value in PCR15 (and the values in other PCRs) would be corrupt. But what if the user hadn't sealed any data?

To address this potential security concern, we propose an additional security feature for attestation by the user: a *catch phrase*. On each update or install, the user enters a string, picture, or other type of data which presumably would be known only to him. We call this string a catch phrase. During the install/update procedure, the catch phrase is sealed by the final value of PCR15 (ie. the hashes of SABLE, Genode, and the replay-prevention key).

So, whenever the user would like to attest to the integrity of Genode, he calls some function which attempts to unseal the catch phrase and print it to the screen. If the system is in a secure state, then the value of PCR15 will be valid, and the unseal will succeed. The user observes the catch phrase and verifies that it was the one he had entered on the last install/update. If the system is not in a secure state, then the unseal will fail, and the correct catch phrase cannot be retrieved.

Printing a secure string to the terminal may at first seem counterproductive and risky. However, the catch phrase can *only* be printed if the trusted Genode was launched properly. So only trusted software could intercept the catch phrase, which we certainly hope that the trusted software will not attempt to do.

Another alternative is to have the catch phrase printed by SABLE at boot. This could happen after the DRTM instruction has run, and before the APs are initialized. So the user may be presented with a prompt such as

```
The retrieved catch phrase is:
    "Go Orange!"
Is this correct (y/n)? y
...
Loading Genode
```

By outputting the catch phrase while only the bootstrap processor is running, we minimize the risk that it will be intercepted. And if the user is never prompted to verify a catch phrase, then he should know that the secure environment was not booted.

Note that we shouldn't lose any security by adding this feature. The user in the scenario above would be oblivious to the fact that he is running a malicious environment. With the catch phrase, the user is presented with unforgeable visual evidence that his system has booted properly.

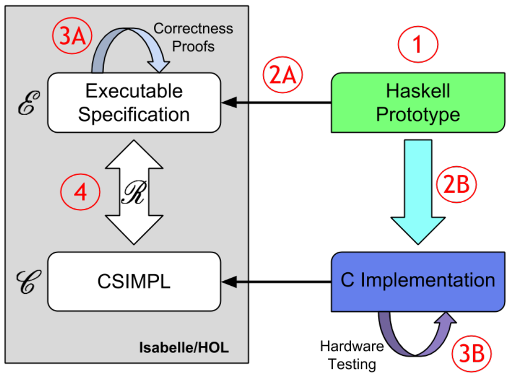

Figure 1: Project flow is as follows: (1) Write and test the Haskell prototype, (2A) team A translates the prototype to Isabelle/HOL and (2B) team B translates the prototype to C, (3A) team A proves correctness properties of the Executable Specification while (3B) team B tests the C implementation on hardware, then parses C into CSIMPL, (4) Refinement $\mathscr{R}$ is established between the Executable Specification $\mathscr{E}$ and the CSIMPL $\mathscr{C}$.

3.4 The Development Process

Figure 1 provides an overview of our design process for the entirety of the project. Once prototyping is complete, the SABLE group will split into two teams, A and B, which will work in parallel on separate but related tasks. Team A will translate the Haskell model into the theorem prover Isabelle/HOL, yielding the executable specification[3]. We plan on utilizing an automatic translation tool, Haskabelle, with manual corrections when needed.

Team B will simultaneously perform a fully manual translation from Haskell to C. After Team A has completed its translation, it can begin devising and proving functionality properties of the loader via Hoare logic. This is accomplished by decomposing the executable specification into function statements, embedding each statement into a Hoare triple, and proving that each precondition implies each post-condition. Only after functionality has been established can we begin to prove security properties[11]. When Team B finishes its translation, it will simulaneously begin testing the loader on raw hardware.

Inevitably, both teams will encounter bugs and design flaws during this stage of the project. Such issues must be resolved by collaboration across the entire research group. The group may have to repeatedly revisit the Haskell prototype, and each team must then redo parts of its translation and verify the outcome until each problem has been resolved.

When the group is satisfied with the correctness proofs and the implementation, the C code will be put through a parser to produce a representation in CSIMPL, an imperative C-like language built into Isabelle/HOL[16].

With all aforementioned tasks complete, we prove refinement. That is, we prove that the behavior of the C implementation is a subset of the behavior of the executable specification. This will likely be the most arduous design phase. For $8,700$ lines of C code, the seL4 developers estimate that refinement at this level took just over 2 person years to complete, with $37,300$ lines of hand-written proof[16]. Fortunately our loader should be much shorter (perhaps 1,000 LoC) and will have an overall lower degree of complexity due in part to its sequential nature. Moreover, refinement is expedited by having programmers working in parallel, with each programmer working on one function at a time. The seL4 group estimates that an average of 3-4 functions were proven per person per week[16]. One option we are currently weighing is to crowdsource the refinement. Since refinement is somewhat mechanical, but not mechanical enough to automate, this may prove an effective strategy for approaching the most tedious stage of development. With refinement established, the C implementation must maintain all of the Hoare logic properties which were proven for the executable specification[8].

In order to ensure that refinement will be possible at all given our design scheme, we will select several representative functions in the Haskell prototype to translate and refine immediately. It is our hope that this feat will provide us with insight into the refinement process. This may help us avoid potentially costly design choices throughout the remainder of the development process.

3.5 Haskell Model

Our present work is focused on completing the Haskell model. One of our goals is to make the model easily adaptable to new architectures. Once we have a final product for the AMD SVM architecture, we will adapt the design to Intel's similar TXT architecture. After that, we will design and implement a different, but also formally verifiable secure loader for ARM TrustZone. We allow for pliancy in the Haskell prototype by maintaining a high level of abstraction. Our strategy is to divide the model into three layers:

- The **machine layer** contains information about a set of architectures. If `TARGET`[5] is some architecture we wish to add to the model, then we create three new modules:
  - `RegisterSet/TARGET.lhs` has, among other features, a data structure describing an enumerated set of registers within this architecture's CPU.
  - `Devices/TARGET.lhs` has a list of devices (ie. the TPM) associated with the architecture. Device-related definitions themselves are located separately, but still within this layer.

[5]`TARGET` is actually a macro which is defined during compile time.

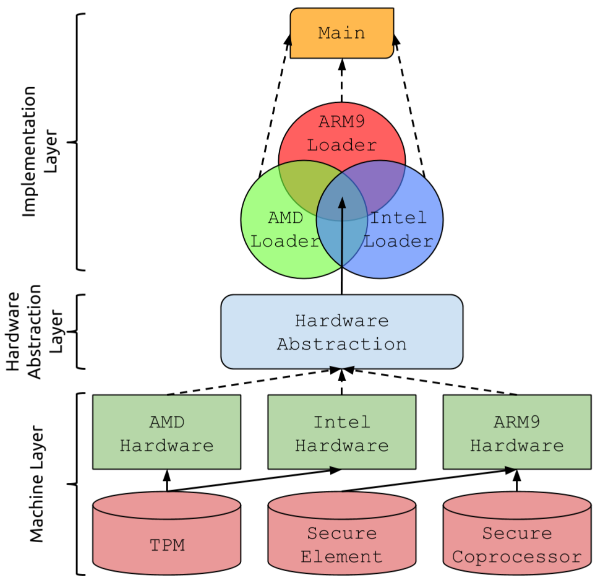

Figure 2: The Haskell Prototype

  – `Hardware/TARGET.lhs` has any other information pertaining to `TARGET` which does not fit into the other two files.

  In addition to these architecture specific modules, the machine layer also contains device modules. For example, the TPM driver is defined in `Devices/TPM.lhs`. This setup allows any architecture which uses a TPM to simply import the TPM module.

- The **hardware abstraction layer** is static. Once the design is perfected, it should not need to be altered to compensate for new architectures or loader features. It abstracts the machine independent features by providing a model of physical memory, system failures, and state-based program execution, as well as an interface to the simulated machine layer.

- The **implementation layer** consists of "do-notation" monadic code, which in essense can be read as imperative code. It is this layer which will ultimately map onto the final C implementation.

Figure 2 depicts this strategy in pictoral form. The machine layer contains architecture specifics as well as device drivers. Each architecture imports only the devices it needs; this is specified for the `TARGET` architecture in the `Devices/TARGET.lhs` file.

The solid arrows in Figure 2 represent fixed imports. For instance, the AMD SVM will always import the TPM, and any model will always import the hardware abstraction layer. The dashed arrows represent variable imports, the variable being the `TARGET` architecture. Any module which imports a variable architecture calls upon the C preprocessor to determine which architecture it should import. So `TARGET` is no more than a macro which is filled in by the programmer at compile time.

The hardware abstraction layer contains only abstract definitions of hardware specifics. For example, many modules in this layer contain signatures of the form

```
import Prototype.Machine.Hardware.TARGET
    as Arch
```

and later contain abstract type definitions such as

```
newtype Register = Register Arch.Register
```

Then, the AMD hardware module would contain a concrete definition of `Register`:

```
data Register = EAX | EBX | ECX | EDX |
    EDI | ESI | EBP | EIP | ESP | CS  |
    DS  | ES  | FS  | GS  | SS
```

This strategy ensures that the hardware abstraction layer will be fully portable, and effortlessly adaptive to new architectures.

After the hardware abstraction layer has collected the `TARGET` architecture specifics, it in turn can be imported into a loader model. These models comprise the implementation layer. Since implementation details such as types, drivers, and runtime helper functions may cross over between architectures, we can and should modularize any commonalities (hence the Venn diagram). Finally, the `Main` function imports the `TARGET` architecture's loader model, and runs the monad.

When making changes to unified portions of the prototype, such as the hardware abstraction layer, it would be convenient to smoothly transition between architectures (ie. for testing purposes). Our scheme makes architecture swaps as easy as a few keystrokes.

4 Discussion

In this section we address some of the claims that were stated in Section 2.2.

4.1 Lightweight / Minimal TCB

A major goal of the x86 DRTM instructions is to allow systems programmers to minimize the TCB[5]. Since trust and security are independent of whatever executes prior to the DRTM instruction, we allow a robust boot loader to perform the grunt work. This entails reading the Master Boot Record (MBR), loading the (encrypted) hypervisor and modules into memory, etc. Such menial and lengthy procedures would be horrendously tedious to verify, if not impossible given the small scope of this project.

The exclusion of any and all unnecessary components from our secure loader makes it, by definition, minimal. We only require the features which assure that our loader is secure, and that the hypervisor is trusted. These features are outlined above in Section 3.2, and mostly involve interaction with the TPM. The code base estimate of 1,000 LoC is based on our analysis of OSLO. OSLO is roughly 1,000 LoC. While it does not provide the same security guarantees as our loader, it also has extra chain-loaded modules. We will not need these. As such, with the exclusion of additional modules, and the inclusion of further security enhancements, 1,000 LoC is a rather sanguine estimate.

4.2 Fast Execution

The TPM is notoriously slow at encrypting large chunks of data[10]. Yet we are forced to use the TPM for encryption, as it is our root of trust. To resolve this conflict we adopt an excellent compromise which maintains security without sacrificing performance.

Passing the entirety of the hypervisor to the TPM for encryption and decryption (for instance, using `TPM_Seal` and `TPM_Unseal`) is something we wish to avoid at all costs. Instead, during the install/update procedure (Section 3.1) we generate a random key to be used for symmetric encryption. A software algorithm then uses the key to encrypt the hypervisor and any related components. Finally the key itself is sealed by the TPM, which will use a negligible amount of clock cycles.

In essence, the TPM is only responsible for the finesse work, sealing the symmetric encryption key to the SRK (Storage Root Key) and the system state, which is held in PCRs. The grunt work of mass encryption/decryption is allotted to the CPU. Decrypting the hypervisor is largely a mirror image of this procedure (Section 3.2).

4.3 Installation and Updates

An outline of the minimal necessary routine is given in Section 3.1. Though the routine itself is systematic and fixed, the environment in which it is executed is debatable. The update/install process is indeed security-critical. A couple of of enticing options include (a) having the hypervisor perform updates automatically in the background, and (b) running each update in a PAL[10], with the user directly calling the update routine. Ultimately this decision will rest with enterprise adopters of SABLE. But they are at least worth our consideration now, for the SABLE team's design decisions will likely affect the feasibility of various install and update procedures.

There is one other matter of interest pertaining to updates. The justly circumspect user will seal his sensitive data using the system state and any other relevant runtime information stored in PCRs. If the core system (ie. Genode) is updated, the hash chain will change, and the user will suddenly find himself unable to unseal his data. To account for this, the user (or hypervisor) may generate symmetric encryption keys, which are in turn used to encrypt/decrypt sensitive data. The TPM then seals these keys to system and/or other program states. Under this paradigm a core system update additionally requires unsealing the keys with the old system measurements, then resealing them with the new measurements. It will again be up to enterprise adopters to implement a secure storage mechanism to hold these keys.

4.4 Nested Boot and Flicker

Two features which may be attractive to enterprise adopters are nested boot and Flicker. In a nested boot scenario, new VMs may be launched within currently existing VMs. To launch securely, these new VMs may again use `skinit`, even after the `skinit` instruction was already fired by SABLE when the system was booted. Flicker, on the other hand, performs secure, isolated computations by calling `skinit` with a Piece of Application Logic (PAL) as its argument[10]. The PAL is a small, trusted piece of code which executes just as the SL would, but with a different purpose. PAL n may use its hash to unseal some sensitive data which had been sealed by PAL $n-1$. PAL n then performs some operations on this data, and again seals it with the hash of PAL $n+1$. In this manner, sensitive data can be directly manipulated without ever being exposed to the currently running execution environment.

The problem with schemes such as nested boot (with a DRTM instruction) and Flicker is that they reset all of the resettable PCRs, namely 17-22. These are our DRTM PCRs, and thus resetting them would cause us to lose our entire boot record. We overcome this restriction by extending hashes of all the necessary boot-related components into PCR15, a non-resettable register. Our choice of PCR15 over one of the other non-resettable PCRs was actually arbitrary; we could have just as well selected any of PCRs 6-14, as these are not implicitely extended during system boot.

With a digest of the boot record in PCR15, high-level application code can then use the value in this register to, for instance, attest to the state of the system or seal/unseal sensitive data. To shield this register from DOS attacks, it is essential that the enterprise adopters do not delegate to VMs the capability to extend PCR15. It is recommended, however, that VMs should be able to extend other non-resettable PCRs (ie. 6-14) to attest to their local state.

4.5 DOS Attacks

The TPM is notorious for its susceptibility to DOS-style attacks. As our design is tethered so intricately with the capabilities of the TPM, the SABLE/Genode framework will inevitably be prone to DOS as well. But this will not undermine security. Additionally, a user can protect himself from much of the inconvenience posed by DOS attacks by safely storing his sealed data on an external device.

DOS attacks can come from two directions: top and bottom. On the bottom end, a malicious BIOS or GRUB could, for instance, tamper with SABLE, which is never sealed. Any such tampering will of course corrupt the value of PCR17, and as a result none of the sealed data will be jeopardized. Yet the user may be unable to load his VM(s). At this stage, he should reboot and load a different setup through GRUB, and then systematically restore the malicious low-level software in his system. After this has been done correctly, he can boot back into SABLE/Genode and unseal his sensitive data. Though this sounds inconvenient, this practice is what separates a secure loader from a trusted loader.

On the top side, DOS attacks could come in numerous forms, many of which would be directed at the TPM. It will ultimately be the job of enterprise adopters to prevent these types attacks through careful access control implementation.

4.6 Replay Attacks

By virtue of our employing TPM sealed storage, we are by default immune to external replay attacks. Consider the following example. Bob seals some sensitive data using the core system state (PCR15) and several other PCRs. He then rationally exports the sealed data to a flash drive. But suppose that an attacker, Alan, steals the flash drive, or else intercepts the sealed data over the web. In addition, Alan steals an event log from Bob's computer. By replaying the event log on his own machine, Alan gets his PCRs to match Bob's at precisely the moment when Bob sealed his data. Fortunately, matching PCR values alone are not enough to breach the security of the TPM. On every `TPM_Seal` call, the TPM encrypts the input not only with PCR values, but also with its private Storage Root Key (SRK). As the SRK is unique to each TPM[14], Alan's replay attack invariably fails.

Local replay attacks also exist, and are also preventable. For instance, we recently "patched" a local replay vulnerability in SABLE. The vulnerability was as follows. Suppose an attacker replays our boot sequence by hashing the exact same boot components and that he has compromised GRUB on our local system. He extends the hash chain into PCR15, and at this point he can hypothetically unseal any data which was sealed using only the system state[6].

The bug was fixed by specifying a new randomly generated replay value, R, which is sealed using the PCR17 value (ie. the hash of SABLE). R is stored in the TPM in such a manner that will only allow it to be read once per boot cycle. This prevents software which executes after SABLE from accessing the replay value. Hence our loader, and only our loader will be able to unseal R and extend it to the bootstrap hash chain.

[6]The observant reader will notice that this example seems to contradict our axiom that we need not trust GRUB under the DRTM model. The counterargument is that PCR15 is *not* in the DRTM model. It is a member of the SRTM model. This attack would be infeasible on the DRTM PCRs, since they must be reset by the DRTM instruction. Even if an attacker does fire the instruction, he must pass in SABLE, or else the value in PCR17 will be corrupt.

4.7 Running the Manual

To simultaneously develop the loader code and its reference manual, we follow the example set by seL4 and adopt Literate Haskell as our prototyping language[4]. Literate Haskell allows a developer to directly embed Haskell code into a document produced by LaTeX. When GHC (the Glasgow Haskell Compiler) compiles the source file, it ignores everything outside of the Haskell code environements (ie. it ignores all of the LaTeX code). When the LaTeX compiler processes the source file, it formats the Haskell code and embeds it in the document. Hence the source code and the manual are one and the same.

References

[1] *AMD64 Virtualization Codenamed "Pacifica" Technology Secure Virtual Machine Architecture Reference Manual.* Advanced Micro Devices, 3.01 edition, May 2005.

[2] David Challener, Kent Yoder, Ryan Catherman, David Safford, and Leendert Van Doorn. *A Practical Guide to Trusted Computing.* IBM Press, first edition, 2007. ISBN 9780132398428.

[3] Kevin Elphinstone, Gerwin Klein, and Rafal Kolanski. Formalising a high-performance microkernel. In Rustan Leino, editor, *Workshop on Verified Software: Theories, Tools, and Experiments (VSTTE 06)*, Microsoft Research Technical Report MSR-TR-2006-117, pages 1–7, Seattle, USA, aug 2006.

[4] Kevin Elphinstone, Gerwin Klein, Philip Derrin, Timothy Roscoe, and Gernot Heiser. Towards a practical, verified kernel. In *Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, HOTOS'07, pages 20:1–20:6, Berkeley, CA, USA, 2007. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1361397.1361417.

[5] Bernhard Kauer. Oslo: improving the security of trusted computing. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, SS'07, pages 16:1–16:9, Berkeley, CA, USA, 2007. USENIX Association. ISBN 111-333-5555-77-9. URL http://dl.acm.org/citation.cfm?id=1362903.1362919.

[6] Gerwin Klein. Operating system verification — an overview. *Sādhanā*, 34(1):27–69, February 2009.

[7] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an os kernel. In *ACM SYMPOSIUM*

ON OPERATING SYSTEMS PRINCIPLES, pages 207–220. ACM, 2009.

[8] Gerwin Klein, Thomas Sewell, and Simon Winwood. Refinement in the formal verification of the sel4 microkernel. In David S. Hardin, editor, *Design and Verification of Microprocessor Systems for High-Assurance Applications*, pages 323–339. Springer US, 2010. ISBN 978-1-4419-1538-2. doi: 10.1007/978-1-4419-1539-9_11. URL http://dx.doi.org/10.1007/978-1-4419-1539-9_11.

[9] J.M. McCune, B. Parno, A. Perrig, M.K. Reiter, and A. Seshadri. Minimal tcb code execution. In *Security and Privacy, 2007. SP '07. IEEE Symposium on*, pages 267–272, 2007. doi: 10.1109/SP.2007.27.

[10] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for tcb minimization, 2008.

[11] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. sel4: from general purpose to a proof of information flow enforcement. In *IEEE Symposium on Security and Privacy*, pages 415–429, San Francisco, CA, May 2013. ISBN 10.1109/SP.2013.35.

[12] Bryan Parno. The trusted platform module (tpm) and sealed storage, June 2007. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.187.2027&rep=rep1&type=pdf.

[13] *TCG Specification Architecture Overview*. TCG, 1.4 edition, August 2007.

[14] Allan Tomlinson. Introduction to the tpm. In *Smart Cards, Tokens, Security and Applications*, pages 155–172. Springer US, 2008. ISBN 977-0-387-72197-2. doi: 10.1007/978-0-387-72198-9_7. URL http://dx.doi.org/10.1007/978-0-387-72198-9_7.

[15] *TCG PC Client Specific TPM Interface Specification (TIS)*. Trusted Computing Group, 1.21.1.00 edition, April 2011.

[16] Simon Winwood, Gerwin Klein, Thomas Sewell, June Andronick, David Cock, and Michael Norrish. Mind the gap a verification framework for low-level c, 2009.