8-2018

Formal Verification of a Modern Boot Loader

Scott D. Constable
Syracuse University, sdconsta@syr.edu

Rob Sutton
RJMetrics

Arash Sahebolamri
Syracuse University, asahebol@syr.edu

Steve Chapin
Syracuse University, chapin@syr.edu

8-2018

Formal Verification of a Modern Boot Loader

Scott D. Constable

Rob Sutton

Arash Sahebolamri

Steve Chapin

Formal Verification of a Modern Boot loader *

Scott Constable Rob Sutton Arash Sahebolamri Steve Chapin
Department of Electrical Engineering and Computer Science
Syracuse University
Syracuse, New York, 13244-1200

Abstract

We introduce the Syracuse Assured Boot Loader Executive (SABLE), a trustworthy secure loader. A trusted boot loader performs a cryptographic measurement (hash) of program code and executes it unconditionally, allowing later-stage software to verify the integrity of the system through local or remote attestation. A secure loader differs from a trusted loader in that it executes subsequent code only if measurements of that code match known-good values. We have applied a rigorous formal verification technique recently demonstrated in practice by NICTA in their verification of the seL4 microkernel. We summarize our design philosophy from a high level and present our formal verification strategy.

1 Introduction

The United States Department of Defense Orange Book defines the Trusted Computing Base (TCB) of a computer system as the part of the computer system "which contains all of the elements of the system responsible for supporting the security policy and supporting the isolation of objects (code and data) on which the protection is based" [15]. Hence, on a typical x86 server the TCB or "trust boundary" would encompass the hardware, firmware, boot loader code, operating system, and web server software. Such TCBs could easily comprise millions of lines of code.

The surface area of the problem may be reduced in two ways. The first and most obvious solution is to shrink the TCB, either by reducing the amount of code required to perform the desired tasks, or reducing the amount of code which needs to be trusted. The latter can be accomplished by utilizing hardware protections such as AMD SVM [5] and Intel TXT [20], which effectively remove pre-operating system software and firmware from the TCB. ARM TrustZone [3] partitions system code into "normal world" and "secure world" code, ideally to exclude the normal world code from the TCB. More recently, Intel SGX technology [2] introduced a hardware-protected execution environment to shield arbitrary code (e.g. system or application code) from the rest of the system.

The second solution is to increase the *trustworthiness* of code which must be a member of the TCB. For instance, a program written in a memory-safe language may be more trustworthy than a similar program written in C. Stronger guarantees about the trustworthiness of a program or system can be made using formal methods. Substantial progress on producing partially or fully verified operating systems, microkernels, and hypervisors has been made in the past decade [17, 23, 27].

SABLE is a trustworthy secure loader which applies both of these solutions to shrink the platform TCB and improve the trustworthiness of platform software. A trusted boot loader performs a cryptographic measurement of program code and then ex-

*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)"

ecutes it unconditionally. Later-stage software may opt to verify the integrity of the system through local or remote attestation. A secure loader differs from a trusted loader in that it executes subsequent code only if measurements of that code match known-good values.

Hence a secure loader must, by definition, prevent the execution of untrusted code [30]. SABLE is able to make this guarantee by utilizing the Trusted Platform Module (TPM) [36] chip together with AMD SVM on AMD platforms and Intel TXT on Intel platforms. Via cryptographic hashing, the TPM can "measure" code prior to its execution. Additionally the TPM may "seal" data to a particular system state, characterized by hash chain digests aggregated in secure storage [30]. By joining these two paradigms, SABLE satisfies the definition of a secure loader.

Moreover we employ formal verification techniques demonstrated in practice during the seL4 verification effort [24]. In particular, we have implemented SABLE in a manner which allows it to be translated into a monadic language that can be parsed by a proof assistant. In this proof assistant, we construct an abstract specification of SABLE's implementation, and prove that the implementation exhibits a subset of the behavior allowed by the abstract specification. This rigorous verification effort thus increases the trustworthiness of SABLE when compared against other trusted software which has only been penetration tested.

We have implemented SABLE to run on both the AMD SVM and Intel TXT architectures. From the user's perspective, the behavior of SABLE on either architecture is identical. Though the implementation details do differ somewhat, for brevity we focus our discussion in this paper on SABLE's implementation for AMD SVM.

The rest of this paper is organized as follows. Section 2 introduces the relevant background in trusted computing and formal verification. Section 3 outlines the design of SABLE and its bilateral attestation protocol. Section 4 describes the implementation of SABLE. Section 5 details the formal methods used to verify SABLE's implementation. Sections 6 and 8 discuss related work and the work remaining to be done to fully formally verify SABLE, respectively. Section 7 discusses our reflection on the design and verification process.

2 Background

2.1 Trusted Computing

According to the Trusted Computing Group (TCG),

> Trust is the expectation that a device will behave in a particular manner for a specific purpose. A trusted platform should provide at least three basic features: protected capabilities, integrity measurement and integrity reporting. [37]

SABLE serves as the software foundation for integrity measurement and utilizes the protected capabilities of the trusted platform. The following subsections introduce these concepts. Integrity reporting may be performed by the operating system or application software which is outside the scope of this paper. Details about integrity reporting with the TPM can be found elsewhere [11, 31, 37, 44].

2.1.1 Integrity Measurement

The TCG uses the term *measurement* to describe a cryptographic hash operation [37]. Measurements can, among other purposes, be used to verify the integrity of code/data or to attest to the integrity of a particular system configuration. TPM chips contain several Platform Configuration Registers (PCRs) which store measurements in a digest. Measurements, however, are not written directly into a PCR. Rather, they are *extended* into a PCR in the following manner:

$$\mathrm{PCR}_{i,n+1} \leftarrow H(\mathrm{PCR}_{i,n} \,||\, H(data))$$

where H is a cryptographic hash function, $||$ is the concatenation operator, and $\mathrm{PCR}_{i,n}$ is the value of the ith PCR after n extend operations on that PCR [37]. Thus in each extend operation the current value in the PCR is replaced by the hash of the old value of the PCR concatenated with the hash of

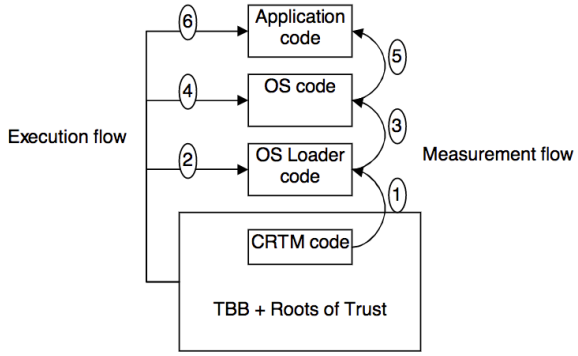

Figure 1: Trusted boot execution [37]

the new data. Because the PCRs are shielded by the TPM and a narrow command interface, they form a *root of trust for measurement* (RTM) for the system.

The PCR extension scheme allows one to chain hashes in a sequence of arbitrary length, e.g. as a digest of system or process execution. The common application of this technique to trusted boot is as follows. Some initial component known as the *core root of trust for measurement* (CRTM) measures itself and the next component to be launched [37]. These measurements are extended into a PCR by the CRTM. The next component in turn measures its subsequent component(s), etc. When the operating system or hypervisor is finally launched, it is then responsible for measuring each program and extending that measurement into a PCR before forking a process for that program. Figure 1 illustrates this procedure. Odd-numbered steps indicate measurements, and even-numbered steps indicate the launch of the next component.

Ideally, the OS/hypervisor would also store a more verbose log of system execution, including the boot components. For example, this log may contain the name of each component/program, its version number, and its measurement. The TCG refers to this kind of log as a *stored measurement log* (SML) [37]. A remote client could verify the integrity of a given server by requesting the server's SML and a signed quote of the server's PCR measurements. The client would first verify that all of the software in the SML is sufficiently trustworthy (i.e. known non-malicious). Then the client would use the SML to compute the hash digest(s) in the same manner and sequence as the server software and the TPM, and compare the result(s) against the server's quoted PCR value(s). Assuming that the PCR values can be transmitted to the client with verifiable integrity[1], these PCR values accurately characterize of the state of the server. Thus the remote client is able to determine whether or not the SML is an honest log of the server's running software and firmware. A Linux platform which uses this logging and reporting scheme was described in [33].

In the TCG terminology, the boot protocol given in Figure 1 uses a *static RTM* (SRTM) [37]. A SRTM begins performing measurements as early in the boot process as possible. For instance, on PC clients the BIOS and firmware serve as the SRTM. The BIOS must measure itself and extend the measurement into PCR0. It must then measure ROM code and configurations, and initial program loader (IPL) code, then extend the measurements into other PCRs [41]. The IPL must then continue to maintain the trust chain.

Unfortunately, low-level software such as the BIOS and UEFI are known to be susceptible to numerous attacks [8, 26, 43], and thus are not infrequently compromised. Moreover these components may be regularly updated, hence their measurements would change over time. This could make it difficult to keep track of which measurements are "good" and which are "bad." Even the SRTM model itself has been successfully compromised [10].

Another relevant concern to security experts is the size of the TCB when the entire boot process is included. If all of the code which is executed after power-on must be measured, then the entire system is the TCB. This is not desirable. To mitigate, the TCG has introduced a second form of RTM: "A *dynamic root of trust for measurement* transitions from an untrusted state to one that is trusted" [37, emphasis added]. In other words, in order to minimize

[1]TPM 1.2 provides the `TPM_Quote` command [40], which signs a set of PCR values using an Attestation Identity Key (AIK) [37]. AIKs use Direct Anonymous Attestation (DAA) [9], which allows the AIK's signature to be validated without revealing the identity of the platform owner.

the TCB, the root of trust for measurement should be established as late in the boot process as possible. This is the goal of the DRTM, also called a late launch [20].

Both AMD and Intel have implemented DRTM support in their recent x86 architectures. AMD's DRTM technology is called Secure Virtual Machine (SVM)[2] [5], and Intel's is Trusted Execution Technology (TXT) [20]. These two implementations 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 `getsec[SENTER]`.

In brief, `skinit` takes as an argument a Secure Loader (SL), and extends its hash into a PCR. This establishes the CRTM. 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, and DMA is disabled [5]. The lone CPU core which remains awake during an `skinit` is referred to as the bootstrap processor (BSP). Whatever code was executed before `skinit` has no bearing on the execution of the system after the `skinit` instruction has been invoked. As a consequence the size of the TCB is substantially reduced.

2.1.2 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 [36].

The EK is generally used for attestation purposes, for instance to sign a report which lists the values currently residing in a set of PCRs. Of greater relevance 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 [36]. Performing a `TPM_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 `TPM_Bind` is further enhanced by `TPM_Seal`. 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 `TPM_Seal` command takes as parameters the data to be encrypted and a set of PCR indices[3] [30]. The command outputs the cyphertext encrypted by the SRK, as well as an integrity-protected list of PCR indices and their corresponding values, as specified during `TPM_Seal`. Together, the cyphertext and PCR list comprise an encrypted blob which may be passed into the `TPM_Unseal` command. The unsealing will succeed if the PCR values given in the integrity-protected list match the actual PCR values at the time `TPM_Unseal` is called [30]. If all of the values match, then the TPM uses its private SRK to decrypt the data.

2.2 Data Refinement

Despite the assurances provided by secure hardware and penetration 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 trustworthiness. Until recently, formal methods proved infeasible in the verification of large and complex programs such as operating systems and hypervisors. An extensive overview of the shortfalls and successes of verification efforts in this area was cataloged by Klein [23].

Yet in recent years, the formal verification of large-scale projects has become not only feasible, but also time and cost efficient. The seL4 team at NICTA prototyped, verified, and implemented a high-performance microkernel in an estimated 20 per-

[2]AMD has recently re-branded SVM as "AMD-V."

[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.

son years [24]. Their final product has been proven void of a number of common bugs and vulnerabilities, such as buffer overflow attacks. Moreover the seL4 microkernel's performance is demonstrably on par with that of several other high-performance microkernels in the L4 family [24].

The high-level verification technique used by the seL4 team is known as data refinement [14, 25], which can be summarized as follows. Define a concrete event-driven system C in terms of its state transitions as it reacts to events, e.g.

$$Step_C :: event \Rightarrow \sigma_C \Rightarrow \sigma_C \ set$$

We allow each state transition to produce a set of output states so that we can model both failure (an empty set) and non-deterministic behavior (a set with cardinality greater than 1). Similarly, define $Step_A$ over states σ_A for an abstract event system A. Finally define a global "observable" state σ_G which ideally should be a generalization of both σ_C and σ_A, such that system initialization and finalization functions

$$Init_S :: \sigma_G \Rightarrow \sigma_S \ set,$$
$$Fin_S :: \sigma_S \Rightarrow \sigma_G$$

respectively, where $S \in \{C, A\}$. To simulate a finite sequence of steps, we define a stepping function:

$$steps\ \delta\ s\ events$$
$$:= \mathsf{foldl}\ (\lambda event\ states.\ (\delta\ event)\ ``\ states)\ s\ events$$

where R " S is the image of S under the relation R. Finally we define a function to execute the system, beginning and ending in global state(s):

$$execution\ S\ s\ events :=$$
$$Fin_S\ ``\ (steps\ Step_S\ (Init_S\ s)\ events)$$

Given the abstract system A and concrete system C, we say that C *refines* A (denoted $A \sqsubseteq C$) when

$$execution\ C\ g\ events \subseteq execution\ A\ g\ events$$

for all initial globals states g and event sequences *events*. Hence, the execution of C (in terms of state transitions) does not exceed the bounds of the execution of A. One useful corollary to data refinement is that any Hoare triple of the form

$$\{\!|P|\!\}\lambda s.\ execution\ S\ s\ events\{\!|Q|\!\}$$

which is valid for A must also be valid for C.

3 SABLE Overview

The primary goal of SABLE is to facilitate mutual trust between the user and his or her system. SABLE establishes this trust by implementing a bilateral attestation protocol, which requires the user and system to exchange secrets during the boot. In order to protect the secrets in memory, the bilateral attestation is performed after the DRTM instruction invocation, and within the trusted execution environment set up by the DRTM instruction.

SABLE's execution proceeds in three stages: pre-launch, launch, and post-launch, where the term "launch" refers to invocation of the DRTM instruction. The pre-launch phase establishes communication with the TPM chip and enables the virtualization capabilities of the CPU required for DRTM. It also stops all application processors, leaving only the primary "bootstrap" processor running.

The launch phase invokes the DRTM instruction. In AMD SVM, the DRTM instruction is `skinit`. In brief, `skinit` performs the following operations:

- Reinitialize the CPU in the same manner as the INIT signal.
- Clear or reset all CPU registers except `EAX` and `EDX` (which hold arguments for `skinit`), `ESP`, and MSRs not related to security.
- Set up protections for the 64KB region containing the secure loader (SL), including protection from direct memory access (DMA), isolation from the PCI configuration space, and isolation from any component (e.g. a graphics card) using GART-translated addresses.
- Transmit the SL image to the TPM, which then hashes the image and extends the hash into PCR17.

- Clear the global interrupt flag, thus disabling all interrupts.

Further details about the behavior of `skinit` can be found in the SVM architecture reference [5].

Each v1.2 TPM chip has at minimum 24 PCRs [38]. PCRs 0-15 are generally used for the SRTM model. When the system is reset, these PCRs are all reset to 0. 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 invoke a DRTM instruction. PCR17 cannot be extended by software; it can only be extended by the CPU, e.g. via `skinit`. Hence a non-zero, non-negative-one value in PCR17 can be taken as evidence that the DRTM instruction was invoked on the system and in the current boot cycle, and by the SL whose measurement matches the value in PCR17.

The post-launch stage proceeds in one of two directions: configuration or secure boot. When performing a secure boot, SABLE does the following:

1. Measure the additional boot modules that were loaded by the generic boot loader that preceded SABLE, and extend them into PCR19.
2. Read a sealed blob from a TPM NVRAM index. The blob contains an encrypted "pass phrase," which can be interpreted by the user as evidence that the boot components are valid.
3. The user is prompted to enter two passwords, one to authorize use of the TPM's storage root key (SRK), and another to authenticate the user[4].
4. Issue a `TPM_Unseal` command to unseal the blob containing the encrypted pass phrase. SABLE provides the measurements in PCRs 17 and 19 as well as the authorization passwords to satisfy the access control policy on the sealed blob.
5. If the `TPM_Unseal` operation succeeds, then the access control policy was satisfied. Display the pass phrase to the user, who is then prompted to confirm that the pass phrase is correct. If the user confirms, proceed to launch the next executable module. Otherwise, the access control policy failed, and the TPM will deliver an error code specifying which criteria (e.g. a bad PCR value) was not satisfied; inform the user of the error and exit.

The configuration procedure is roughly the mirror image of the secure boot. The user enters the necessary passwords and creates a pass phrase for the given sequence of modules. SABLE measures the modules and seals the pass phrase to these measurements and the measurement of SABLE itself, then finally stores the sealed blob into a fresh TPM NVRAM index. We used the term SABLE-Enabled Configuration (SEC) to refer to a given sequence of modules and a NVRAM index containing a secret bound to those modules. A typical TPM 1.2 chip with 6KB of NVRAM can support up to 18 SECs.

4 Implementation

Our approach to implementing SABLE was motivated by our intention to formally verify the implementation after development, and after testing SABLE on hardware. We did our original development testing of SABLE in the QEMU x86 CPU emulator, together with a software TPM emulator. We modified QEMU to emulate the `skinit` instruction. We later continued development on an AMD laptop with SVM and a TPM 1.2, and then on an Intel laptop with TXT and TPM 1.2.

In general, formally verifying a low-level language like C is difficult. Unsafe casting, bit fields, unions, heap memory, pointer arithmetic, inline assembly, and other similar features are difficult to model and verify without subjecting the input to some reasonable constraints. For this reason, most of the (very few) existing tools that can be used to verify C code require that some proper subset of C be used.

The tool chain which we use to verify C code is built on top of the Isabelle/HOL proof assistant. Through a multi-step process, the tool chain translates the C input into another imperative language which can be understood by the proof assistant, and then abstracts this respresentation into a state

[4]The use of an additional password for the user is necessary in the common scenario where several users may share access to the SRK.

monad. The state monad representation allows us to model imperative (stateful) programs in a pure functional (stateless) proof assistant language. More details on this process are given in Section 5.

This tool chain does enforce several constraints on its input. For example, it will refuse to process string literals. The obvious workaround was to use the C preprocessor to strip away string literals before passing the source code into the verification tool. The most limiting constraint of the tool chain is that it mandates that local variables be local in a literal sense. That is, local variables may not be passed by reference (e.g. with an `&`) as an argument to a function. Without this constraint, the verification tool chain would have to treat local variables as part of the program state. This in turn would make it difficult or impossible to practically model and verify recursive functions, for instance. Moreover, by treating local variables as strictly local, the they can be safely abstracted into lambda-bound variables, which are much easier to reason about than static or heap data.

In situations where pass-by-reference semantics would normally be desirable, we must instead adopt one of two alternatives. The first alternative is to pass locals by value. However, if the variables are large objects or buffers, this can cause computational overhead. To make matters more difficult, SABLE must fit entirely within a 64KB region of RAM. Minus the code, heap, and data regions, we are left with only about 8KB of RAM for the stack. So having multiple copies of the same object simultaneously on the stack should be avoided. And if an argument to a function should be modified, this copy-by-value scheme will not work, and the function may need to be substantially refactored.

The second alternative is to use pass-by-value semantics with (pointers to) heap-allocated data or statically allocated data. In general, we use calls to `alloc()` only when the size of a buffer cannot be determined at compile time, e.g. when processing responses from the TPM. We statically allocate data which could reasonably be considered a part of SABLE's state, rather than a temporary value.

The structure of the AutoCorres monad also forced us to reconsider the way in which we handle hardware errors and software exceptions. AutoCorres uses the type ('a, 's) nd-monad to model stateful computations. This is a synonym for the expanded type 's ⇒ ('a × 's) set × bool, where × is the Cartesian product of types. That is, an AutoCorres monadic computation takes an input state, and returns a set of return value and result state pairs; hence monadic computations can be non-deterministic. The Boolean is a flag which, when set, indicates that a catastrophic error—one which should never be triggered—has occurred, such as a failed assertion or a null-pointer dereference. The monad can also model exception throwing and handling with the addition of a sum type: ('e + 'a, 's) nd-monad. These computations may yield either an exception of type 'e or a result of type 'a. However, at the time of this writing the AutoCorres tool cannot generate an abstraction of C code to use this exception monad.

SABLE interfaces with multiple hardware components during its execution, including the TPM, CPU, keyboard, display port, and PCI configuration space. Any of these components may fail or emit error codes, from which SABLE cannot always recover. Hence SABLE should be allowed to fail gracefully by simply sending the shutdown signal to the CPU when such an error occurs. Unfortunately, there is no trivial way to model this behavior with the given ('a, 's) nd-monad construct. The failure flag is reserved only for catastrophic failures, which we must prove the absence of. Hardware errors, on the other hand, must be anticipated and either handled (e.g. by retrying the operation) or should trigger a graceful shutdown.

Our approach was to implement SABLE to use return-style exceptions, with considerable help from the C preprocessor. For example, in Listing 1 we have a function, `read_passphrase()`, taken verbatim from the SABLE source code. The `RESULT_(T)` is a preprocessor-facilitated type constructor which, given some type `T`, yields a type which can hold either an exception or a value of type `T`. So `read_passphrase()` can either return (throw) an exception, or a value of type `TPM_STORED_DATA12`. The driver function `TPM_NV_ReadValue()` can also throw exceptions. When a function makes a call to a callee and the callee throws an exception, the caller is

```
1  static RESULT_(TPM_STORED_DATA12) read_passphrase(UINT32 index) {
2    const OPTION(TPM_AUTHDATA) nv_auth = {.hasValue = false};
3    RESULT_(HEAP_DATA) val =
4        TPM_NV_ReadValue(index, 0, 400, nv_auth, NULL);
5    THROW_TYPE(RESULT_(TPM_STORED_DATA12), val.exception);
6
7    return (RESULT_(TPM_STORED_DATA12)){
8        .exception.error = NONE,
9        .value = unpack_TPM_STORED_DATA12(val.value.data,
10                                         val.value.dataSize)};
11 }
```

Listing 1: C implementation of `read_passphrase()`

obliged by convention to either catch (and optionally handle) the exception, or throw the exception to its own caller. This latter behavior is typical in SABLE, because most exceptions stem from hardware issues, and thus must be fatal. The function in Figure 2 does not know how to handle any exceptions potentially thrown by `TPM_NV_ReadValue()`, so it simply throws the exception to its caller. The `THROW_TYPE(T, e)` macro returns an exception of type `T` if `e` is an exception. If the call to `TPM_NV_ReadValue()` succeeds, then `read_passphrase()` returns the null exception `NONE`, with the unpacked data structure containing the passphrase data that was read from the TPM's non-volatile memory.

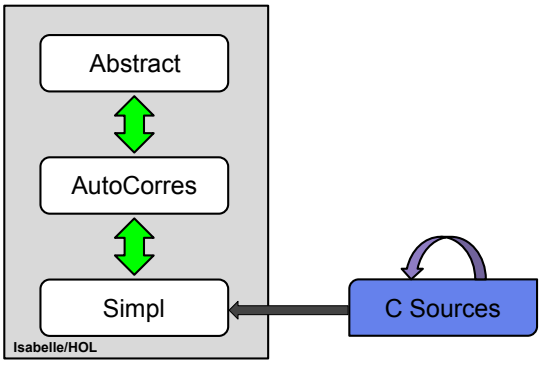

Figure 2: SABLE verification development cycle

At the root of SABLE's call graph is a function which uses a `CATCH_ANY()` macro to catch any exception which was thrown and uncaught during execution. It is this function which performs the graceful shutdown in the case of an uncaught exception. This design also granted us one additional noteworthy benefit. Because SABLE runs on bare metal, there is no operating system underneath to produce and pretty-print stack traces in the event of a catastrophic failure. Our `THROW()` macros build a stack trace by recording source code information at each level of the call graph into a linked list. If the root function catches an exception, it then pretty-prints the call stack before the graceful shutdown. We often found this feature to be useful for debugging.

5 Formal Verification

All of our formal verification has been done in the Isabelle/HOL proof assistant. Isabelle's efficacy in software verification has already been demonstrated by the seL4 team [24]. Our verification process follows a strategy based on their approach.

Figure 2 provides an overview of the SABLE development and verification process. We developed SABLE in an iterative process, gradually adding new features to the C implementation and penetration testing them. The effort to formally verify SABLE is shown on the left side of the figure. White boxes indicate layers of abstraction, and the green arrows are correspondence proofs.

```
definition
  NV_ReadValue :: "NV_ReadValue_in ⇒
      ('a NV_ReadValue_out) tpm_monad"
where
  "NV_ReadValue com ≡ unknown"
```

Listing 2: Under-specified abstract specification of `TPM_NV_ReadValue`

```
definition
  read_passphrase ::
    "nat ⇒
       (string TPM.STORED_DATA) E_monad"
where
  "read_passphrase i
      ≡ TPM_NV_ReadValue i 0 None"
```

Listing 3: Abstract spec of `read_passphrase()`

We use a parsing tool to translate the SABLE source code into Simpl [34], an imperative programming language built into Isabelle/HOL. We then use the AutoCorres tool [18] to automatically abstract the Simpl code into a monadic representation with an abstracted heap model [19]. Moreover, the AutoCorres tool automatically verifies the translation by generating correspondence proofs for each translated function, over each step of the translation process.

We produce the abstract, behavioral specification of SABLE one function at a time. The abstract specification is minimal in the sense that it captures only enough detail about SABLE's behavior that is required in order to describe the security properties we later hope to prove about SABLE. It is also nondeterministic, in that each abstract function may transform one state and one sequence of inputs into a set of (state × output) pairs. The proofs of correspondence between abstract specification functions and AutoCorres-generated functions must be written manually in the proof assistant.

5.1 Modeling SABLE

We use under-specification to model the low-level operations performed by our TPM driver and the TPM itself. Listing 2 shows the abstract specification of `TPM_NV_ReadValue`, the command which tells the TPM to read data from a specified index in TPM NVRAM. NV_ReadValue_in and NV_ReadValue_out are record-style structures which describe the input arguments and output values, respectively, to the command. These roughly correspond to the command and response values described in the TPM specification [40], but with low-level details, such as parameter sizes, abstracted away.

The TPM itself is modeled with full nondeterminism. That is, the TPM's state is not explicitly modeled; nor are its commands. The tpm_monad uses a unit (empty) state and an exception type lifted from the TPM specification. When the TPM driver invokes a TPM command, the command is lifted into the caller's monad, which does model CPU and memory state. This separation implies that TPM operations cannot affect the state of the CPU and memory (including SABLE), except for the value(s) returned by the operation. The unknown value in Listing 2 models full non-determinism by returning the universal set of values for the inferred type. In this case, TPM.NV_ReadValue will return an arbitrary value of type ('a NV_ReadValue_out), where 'a is the type of the value to be read, e.g. a string. Or, TPM.NV_ReadValue could return an arbitrary TPM error, since tpm_monad also models exceptions.

Functions in SABLE are modeled with greater determinism. The abstract specification of the SABLE function which reads the passphrase from TPM NVRAM is shown in Listing 3. It simply calls the TPM_NV_ReadValue function, which is the abstract representation of our C API function of the same name. It takes the function parameters and marshals them into a buffer that is transmitted to the TPM, then processes the response buffer and unmarshals the results, returning them to the caller. The C implementation of read_passphrase needs to perform additional unmarshalling on the data returned from the API call, but this modeling technique allows us to abstract away these mundane details. The next section explains why this simplification does not violate correspondence between the layers of abstraction.

$$
\begin{aligned}
&\mathsf{corres}\ R_S\ R_R\ P\ P' \ \equiv \\
&\quad \lambda m\ m'.\, \forall (s, s') \in R_S.\, P\ s\ \wedge\ P'\ s' \\
&\qquad \longrightarrow (\forall (r', t') \in \mathsf{mResult}\ (m'\ s').\ \\
&\qquad\quad \exists (r, t) \in \mathsf{mResult}\ (m\ s).\ \\
&\qquad\qquad (t, t') \in R_S\ \wedge\ R_R\ r\ (r', t')) \\
&\qquad \wedge\ \neg\, \mathsf{mFail}\ (m'\ s')
\end{aligned}
$$

Figure 3: Correspondence definition in Isabelle/HOL

$$
\frac{\begin{array}{c}\{\!|P|\!\}\ a\ \{\!|Q|\!\} \qquad \{\!|P'|\!\}\ c\ \{\!|Q'|\!\} \\ \mathsf{corres}\ R_S\ R'_R\ P\ P'\ a\ c \\ \bigwedge r\ r'.\, \mathsf{corres}\ R_S\ R_R \\ \qquad (Q\ r)\ (\lambda s'.\, r'\ r\ (r', s') \wedge Q'\ r'\ s')\ (b\ r)\ (d\ r')\end{array}}{\mathsf{corres}\ R_S\ R_R\ P\ P'\ (a \gg\!= (\lambda r.\, b\ r))\ (c \gg\!= (\lambda r'.\, d r'))}
$$

Figure 4: Correspondence splitting rule

5.2 Correspondence

For each function in the abstract specification, we prove that it *corresponds* to the AutoCorres-abstracted implementation function which it is supposed to model. Correspondence is a concept that was first applied to verify seL4 [25]. It is similar to the technique of forward simulation [14], except that it can be recursively applied over monads using a splitting rule. Both correspondence and forward simulation imply data refinement [25].

Our definition of correspondence is similar to the definitions used to verify seL4. It establishes a relationship between monadic functions in our abstract specification and monadic functions generated by AutoCorres. The definition is given in Figure 3.

The m and m' functions are corresponding monads in the abstract spec and AutoCorres output, respectively. R_S is a relation defined over abstract and AutoCorres states and R_R is a relation defined over the monads' return values. P and P' are preconditions which may constrain the input states and/or monadic function arguments. Hence, the definition states that for all input states s and s' which satisfy the state relation and the preconditions, the following hold. First, for each possible output pair of return value r' and state t' from the computation $m'\ s'$, there exists a corresponding output pair r and t from the computation $m\ s$ such that the state relation holds for t and t' and the return relation holds for r and the pair (r', t'). Second, the AutoCorres computation must not fail. The return relation must also accommodate the AutoCorres state because a return value r' may involve a pointer to a value on the heap or in static memory; these are captured as part of the AutoCorres state t'. In our abstract specification, there is no concept of a pointer.

All of our correspondence proofs follow a simple pattern. Given an AutoCorres-generated compound statement $c \gg\!= d$ and a compound statement $a \gg\!= b$ in the abstract specification, we apply a splitting rule (Figure 4) which allows us to prove correspondence separately for c and a, and then for d and b. The $\gg\!=$ operator is a monadic bind for ('e + 'a, 's) nd-monad, which we described in Section 4.

Given abstract and AutoCorres-generated functions f_A and f_{AC}, respectively, we recursively apply this splitting process to their ASTs, until we are left with only atomic statements or nullary return statements, similar to no-ops. At this point, we unfold the definition of correspondence for each pair of atomic or nullary return statements, and use the proof assistant's built-in theorem provers[5] to automatically discharge the proof goal. The most difficult part of this entire process is choosing the correct preconditions P and P' for each splitting step.

Lemma 1 illustrates the typical format for one of our correspondence proofs. This lemma establishes correspondence between the abstract specification of `read_passphrase()` shown in Listing 3 and AutoCorres-generated representation of the C implementation given in Listing 1:

Lemma 1. *The abstract function* read_passphrase *corresponds to the generated* read_passphrase'*:*

$$
\begin{aligned}
&\forall i.\, \mathsf{corres}\ (\mathsf{R\_STORED\_DATA\_rel}\ \mathsf{string\_rel})\ \top\ \top \\
&\quad (\mathsf{read\_passphrase}\ i)\ (\mathsf{read\_passphrase'}\ (\mathsf{of\_nat}\ i))
\end{aligned}
$$

Proof. By recursively applying the correspondence splitting rule, and the definition of correspondence. □

[5] We use the Z3 [13] and CVC4 [7] theorem provers.

The R_STORED_DATA_rel is a type-parameterized relation defined over data of variable size stored on the heap, in this case a string. Counterintuitively, the input states and data do not require any constraints. One might expect that the NVRAM index i should be bounded according to the size of the TPM's NVRAM. We do not enforce this constraint because the value of i is configured by the user, and thus may be invalid. If the value of i is invalid, the TPM will return an error. Thus our proof must demonstrate that both abstraction layers will equivalently handle this error if it is issued by the TPM. In other words, the precondition constraints must only preclude states and/or input values which the functions in question should not be expected to handle.

The correspondence splitting rule does not always precisely fit a desired proof goal. For example, the monadic definition of read_passphrase is not compound (there is no $\gg\!=$ in the definition). However, the AutoCorres-generated read_passphrase' is compound. To apply the splitting rule, we first apply one of the monadic identities to expand the non-compound expression into one that is compound:

$$f\ v \equiv (\mathsf{return}\ v) \gg\!= f \tag{1}$$
$$f \equiv f \gg\!= \mathsf{return} \tag{2}$$

In this example, we use the second identity rule. The return corresponds to an additional unmarshalling operation performed by the read_passphrase'.

The top-level lemma for SABLE proves correspondence for `trusted_boot()`, the function which manages the bilateral attestation described in Section 3. It directly calls `read_passphrase()`, among other routines:

Lemma 2. *The abstract function* trusted_boot *corresponds to the generated* trusted_boot'*:*

$$\forall i.\ \mathsf{corres}\ (\lambda r\ (r', t').\ \mathsf{RESULT\_rel}\ r\ r')\ \top\ \top \\ (\mathsf{trusted\_boot}\ i)\ (\mathsf{trusted\_boot'}\ (\mathsf{of\_nat}\ i))$$

6 Related Work

The first boot loader to implement the TCG platform was Trusted GRUB [1]. Trusted GRUB is a patch added to GRUB which hashes both GRUB and the boot modules loaded by GRUB, and extends them into one or more PCRs. Because Trusted GRUB is built on top of GRUB, it has the advantage of being able to load a wide variety of operating systems and other system software. However, it is also encumbered by a fairly large TCB: the entire GRUB boot loader.

The Open Secure LOader (OSLO) [22] attempted to rectify this issue by using DRTM with a minimal SL. Unlike Trusted GRUB, OSLO does not literally load boot modules from disk. Instead, OSLO is itself loaded as a boot module (e.g. by GRUB). When OSLO is launched it invokes the `skinit` instruction to reset the CPU state, then it hashes the boot modules. Despite having "secure" in its name, OSLO is in fact only a trusted loader because it will unconditionally launch the hypervisor or OS. It does not validate any of the PCR values after measurements after been taken.

The Trusted Boot (tboot) [4] loader, despite having "trusted" in its name, is in fact a secure loader. Unlike OSLO, tboot supports the Intel TXT platform. It achieves secure boot using a launch control policy (LCP)[6] [29], which allows execution to continue after DRTM invocation if and only if the PCR values match the known good values specified by the launch control policy.

None of the aforementioned loaders has been formally verified to any extent. To the authors' knowledge, no other loader has been formally verified to establish security properties. Das Barman and Mukhopadhyay used a model checker to verify a communication protocol between two loaders used in the A380 airplane. But their analysis was focused on safety, not security [12].

The SABLE formal verification effort follows a similar process to that of the seL4 microkernel verification [24], with two noteworthy distinctions. First, the seL4 team began by building a model "executable specification" of seL4 in the Haskell programming language, and testing it on a CPU emulator [16]. Second, the manual seL4 correspondence and refinement proofs extend all the way down to the implementation

[6] LCP is a feature that is only available on Intel TXT.

level. In verifying SABLE, we were able to use the newer AutoCorres tool to first produce a monadic abstraction of the implementation, with automatically generated correspondence proofs.

One other noteworthy project is Flicker [28]. Flicker is a minimal TCB which uses a DRTM instruction to allow user space processes to temporarily construct a trusted execution environment called a Piece of Application Logic (PAL). Instead of invoking the DRTM instruction during boot, Flicker invokes the instruction on demand from the operating system, and on behalf of an application. The code running within the PAL is effectively given the same hardware protections as SABLE.

7 Discussion

7.1 Minimizing the TCB

The primary motivation of the x86 DRTM model is to allow systems programmers to reduce the size of the TCB[22]. Since trust and security do not depend on whatever has executed prior to the DRTM instruction, we allow a generic and versatile boot loader (e.g. GRUB or GRUB2) to perform most of the actual boot loading. This entails reading the Master Boot Record (MBR), loading the hypervisor and modules into memory, and building metadata structures for the loaded boot components. Many of the details of boot loading are complex and low-level, and thus not easily amenable to formal verification. Excluding this code from SABLE and our TCB was thus desirable.

The exclusion of any and all unnecessary components from our secure loader makes it, by definition, minimal. We only require the features which ensure that our loader is secure, and that the operating system's integrity can be verified. These features are outlined above in Section 3, and they mostly involve interaction with the TPM. SABLE's implementation currently contains just under 4,000 (LoC). By contrast, as of this writing GRUB2 contains roughly 300,000 LoC. Thus by adding SABLE to the TCB and removing GRUB2 from the TCB–and other components like the BIOS–we do substantially reduce the size of the TCB.

7.2 Verification Effort

Thus far, we have successfully proved correspondence for the most high-level functions in SABLE's call graph, those which manage SABLE's bilateral attestation protocol. This includes the `read_passphrase()` funtion describd in Section 4, and the `trusted_boot()` function, which is the root of SABLE's call graph after the DRTM instruction has been invoked. A sizable portion of the work to verify correspondence for these functions was dedicated to writing relations between types (including TPM structures) used by SABLE, and abstract representations of those types expressed in Isabelle/HOL. We also found it helpful to prove several lemmas to sidestep boilerplate reasoning about those relations. We have so far written 29 such relations and 13 lemmas about them. Since many other functions in SABLE use these same types, we expect the required effort for the remaining functions to be less than it was for the functions already verified (e.g. in terms of person hours per LoC).

The verification effort has so far entailed more than 400 lines of proof (LoP), and has not yet revealed any bugs or vulnerabilities. However, it has prompted us to refactor or rewrite each of the C functions we have verified to clarify their behavior and make their implementation more concise. For instance, prior to the verification effort we would often declare a structure, and then initialize its fields one-by-one. This increased the complexity of the AutoCorres output, and thus required us to make more splits in our correspondence proofs than were actually necessary. An elegant alternative was to use the C99 [21] designated initializer and compound literal features, which initialize a (possibly temporary) structure with its declaration. The NICTA C parser is aware of these constructs, and AutoCorres treats them as ordinary type constructors. We use both of these techniques in `read_passphrase()`, as shown in Listing 1.

8 Future Work

SABLE itself has no interesting security properties. SABLE merely exchanges information with the user,

and then interfaces with the TPM. This sequence of steps and their significant aspects are captured by the abstract specification, which is in turn refined by the SABLE source code. However, if we reason about SABLE in conjunction with the TPM, the CPU, and the PCI configuration space, then we can begin to formulate meaningful security properties of the system. We are currently planning to build a rudimentary model of the TPM, sufficient to capture the functionality required by SABLE. Only then can we state and prove formal security properties, such as "SABLE will allow the boot to proceed only if the measured boot components have the correct SHA1 hashes, and the user enters the correct authorization data."

It may seem counterintuitive for us to focus lastly on this verification step. But empirically the seL4 team has demonstrated that proving correspondence is far more effective at identifying bugs and vulnerabilities than an analysis specifically focused on abstract security properties. For example, the seL4 correspondence and refinement proofs revealed 150 bugs in their abstract specification, and 144 bugs in their C source code [24]. The proofs that seL4 enforces integrity and authority confinement prompted no further revisions to either the abstract model or the C source [35]. Our plan going forward is to work downwards through the call graph, until we reach functions that are too dependent on hardware to reason about, without modeling that hardware. We will simply assume correspondence for these functions. With these correspondence proofs complete, we will be able to prove the master theorem as a corollary:

Theorem. *SABLE's implementation C refines its abstract specification A:*

$$A \sqsubseteq C$$

The TPM 1.2 specification [39] on which SABLE is currently based has been superseded by the TPM 2.0 specification [42]. We are in the process of updating SABLE with a TPM 2.0 driver and new functionality to take advantage of the improved security capabilities of TPM 2.0. One disadvantage of the SABLE design for TPM 1.2 is that the access control policy for the SEC secrets is extremely brittle. Any update to any of the boot components thus requires the SEC to be completely reconfigured. This is not specifically a limitation of SABLE; it is a limitation of the access control policies that can be applied to sealed data on TPM 1.2. The TPM 2.0 introduces flexible security policies [6], which allow an authorized user to issue updates to the security policy of a TPM entity such as a sealed data blob or NVRAM index. SABLE on TPM 2.0 will allow the SEC owner to sign updated PCR values and produce an authorization ticket which can vouch for the validity of the updated values. The TPM can then validate the ticket, and approve an unseal operation for the new PCR values.

Verification is also posing an additional challenge for our effort to port SABLE to TPM 2.0. Our tentative redesign will no longer use TPM NVRAM to store the SEC secrets. Instead, they will be passed in encrypted form to SABLE as a boot module. Much of the verification that we have completed for SABLE on TPM 1.2 focused on the routines which handle NVRAM storage. These proofs required roughly 50-100 person hours to complete. None of these proofs can be reused for our new implementation. Thus from a formal verification standpoint—with manually written proofs—the cost of a design change is very high.

We are also currently working to formally verify SABLE's heap allocator. This is being done as a separate and independent project [32], which could eventually be used by other trusted systems software. Our goal is to first prove that, for a call to `alloc()` which returns a non-null pointer, the returned pointer references a memory region which (a) is within the bounds of the heap, (b) does not overlap another live memory region on the heap, and (c) is correctly aligned for the type of the object(s) whose allocation was requested. Our second goal is to build a separation logic-based interface in Isabelle/HOL on top of the `alloc()` and `free()` functions, as abstracted by AutoCorres. This interface would allow a proof engineer to use a verification condition generator to automatically discharge proof goals involving calls to `alloc()` and `free()`.

The effort to verify the heap allocator has already revealed two bugs in our implementation. Our formula to convert bytes to heap blocks was suscepti-

ble to unsigned integer overflow. In this case, the heap might allocate fewer bytes than what was requested by the caller. The second bug was a loop termination condition which used a `<` instead of a `<=` to compare two values. Consequently, `alloc()` could allocate memory beyond the end of the heap.

One more feature we are currently working to implement is to allow SABLE to generate a full disk encryption (FDE) key. This will allow SABLE to be a secure loader in the most literal sense of the definition. With this extension, SABLE will use a combination of the pass phrase and a user counter sign (another password) to generate the FDE key, e.g. using a standard cryptographic key derivation function. Thus each SEC would guard a disk partition by encrypting it using the FDE key. The guarded partition could only be unlocked by satisfying the security policy of the SEC through SABLE.

9 Conclusion

In this paper we presented SABLE, a modern boot loader designed to be secure and formally verifiable. SABLE uses a bilateral attestation protocol between the user and the TPM chip to establish mutual trust. Our implementation can operate on both the AMD SVM and Intel TXT architectures. SABLE was also designed and implemented to be amenable to formal verification with the aid of a proof assistant, Isabelle/HOL. The constraints on program input mandated by the verification tool chain required us to employ some unconventional strategies when we implemented SABLE. Finally, we discussed our divide-and-conquer approach to verifying SABLE using a technique to establish correspondence between an abstract model of SABLE and the C implementation of SABLE. The verification effort is extremely labor intensive; we have completed several correspondence proofs, and hope to complete the remainder of them in the near future.

Acknowledgments

Something for CTI and/or CASE? Or for NICTA?

References

[1] GRUB TCG patch to support trusted boot. [Online]. Available: `http://trousers.sourceforge.net/grub.html`. Accessed: 1 August 2016.

[2] Intel® Software Guard Extensions (Intel® SGX). [Online]. Available: `https://software.intel.com/en-us/sgx/details`. Accessed: 9 June 2018.

[3] Security on ARM TrustZone. [Online]. Available: `https://www.arm.com/products/security-on-arm/trustzone`. Accessed: 9 June 2018.

[4] Trusted boot (tboot). [Online]. Available: `https://sourceforge.net/projects/tboot/`. Accessed: 10 September 2017.

[5] Advanced Micro Devices. *AMD64 Virtualization Codenamed "Pacifica" Technology Secure Virtual Machine Architecture Reference Manual*, May 2005. Publication No. 33047, Revision 3.01.

[6] Will Arthur and David Challener. *A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security*. Apress, Berkely, CA, USA, 1st edition, 2015.

[7] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 171–177, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[8] Oleksandr Bazhaniuk, Yuriy Bulygin, Andrew Furtak, Mikhail Gorobets, John Loucaides, Alex Matrosov, and Mickey Shkatov. Attacking and Defending BIOS in 2015. in *ReCon*, Montreal, Canada, June 2015.

[9] Ernie Brickell, Jan Camenisch, and Liqun Chen. Direct anonymous attestation. In *Proceedings of the 11th ACM Conference on Computer and*

Communications Security, CCS '04, pages 132–145, New York, NY, USA, 2004. ACM.

[10] John Butterworth, Corey Kallenberg, Xeno Kovah, and Amy Herzog. Problems with the Static Root of Trust for Measurement. in *BlackHat*, Las Vegas, Nevada, July 2013.

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

[12] Kuntal Das Barman and Debapriyay Mukhopadhyay. Model checking in practice: Analysis of generic bootloader using spin. In Michael Butler, Michael G. Hinchey, and María M. Larrondo-Petrie, editors, *Formal Methods and Software Engineering*, pages 232–245, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[13] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[14] Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1998.

[15] Department of Defense Computer Security Center. Trusted Computer System Evaluation Criteria (Orange Book). Technical Report 5200.28-STD, United States Department of Defense, December 1985.

[16] Philip Derrin, Kevin Elphinstone, Gerwin Klein, David Cock, and Manuel M. T. Chakravarty. Running the manual: An approach to high-assurance microkernel development. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*, Haskell '06, pages 60–71, New York, NY, USA, 2006. ACM.

[17] Jason Franklin, Arvind Seshadri, Ning Qu, Sagar Chaki, and Anupam Datta. Attacking, repairing, and verifying secvisor: A retrospective on the security of a hypervisor. Technical Report CMU-CyLab-08-008, Carnegie Mellon University CyLab, 2008.

[18] David Greenaway, June Andronick, and Gerwin Klein. Bridging the gap: Automatic verified abstraction of c. In Lennart Beringer and Amy Felty, editors, *Interactive Theorem Proving*, pages 99–115, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[19] David Greenaway, Japheth Lim, June Andronick, and Gerwin Klein. Don't sweat the small stuff: Formal verification of c code without the pain. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 429–439, New York, NY, USA, 2014. ACM.

[20] Intel Corporation. *Intel® Trusted Execution Technology (Intel® TXT) Software Development Guide*, November 2017. Revision 015.

[21] ISO/IEC. ISO/IEC 9899:1999 programming languages – C. Technical report, International Organization for Standardization (ISO), Geneva, Switzerland, December 1999. Retrieved from `https://www.iso.org/standard/29237.html`.

[22] 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.

[23] Gerwin Klein. Operating system verification—an overview. *Sadhana*, 34(1):27–69, Feb 2009.

[24] 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 *Proceedings of the*

ACM SIGOPS 22Nd Symposium on Operating Systems Principles, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[25] 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, Boston, MA, 2010.

[26] Xeno Kovah and Corey Kallenberg. How Many Million BIOSes Would you Like to Infect? in *CanSecWest*, Vancouver, British Columbia, March 2015.

[27] Dirk Leinenbach and Thomas Santen. Verifying the microsoft hyper-v hypervisor with vcc. In Ana Cavalcanti and Dennis R. Dams, editors, *FM 2009: Formal Methods*, pages 806–809, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[28] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for tcb minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 315–328, New York, NY, USA, 2008. ACM.

[29] David Mulnix. Intel® Trusted Execution Technology (Intel® TXT) enabling guide. [Online]. Available: `https://software.intel.com/en-us/articles/intel-trusted-execution-technology-intel-txt-enabling-guide`, March 2014. Accessed: 10 June 2018.

[30] Bryan Parno. The Trusted Platform Module (TPM) and sealed storage. unpublished, June 2007.

[31] Mark Ryan. Introduction to the TPM 1.2. unpublished draft, March 2009.

[32] Arash Sahebolamri, Steve J. Chapin, and Scott D. Constable. A formally verified heap allocator. Electrical Engineering and Computer Science Technical Reports 182, August 2018. `https://surface.syr.edu/eecs_techreports/182/`.

[33] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and implementation of a tcg-based integrity measurement architecture. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, Berkeley, CA, USA, 2004. USENIX Association.

[34] Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technical University of Munchen, October 2005.

[35] Thomas Sewell, Simon Winwood, Peter Gammie, Toby Murray, June Andronick, and Gerwin Klein. sel4 enforces integrity. In Marko van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *Interactive Theorem Proving*, pages 325–340, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[36] Allan Tomlinson. Introduction to the TPM. In *Smart Cards, Tokens, Security and Applications*, chapter 7, pages 155–172. Springer US, Boston, MA, 2008.

[37] Trusted Computing Group. *TCG Specification Overview*, August 2007. Revision 1.4.

[38] Trusted Computing Group. *TCG PC Client Specific TPM Interface Specification (TIS)*, April 2011. Specification Version 1.21 Revision 1.00.

[39] Trusted Computing Group. *TPM Main Part 1 Design Principles*, March 2011. Specification Version 1.2, Level 2 Revision 116.

[40] Trusted Computing Group. *TPM Main Part 3 Commands*, March 2011. Specification Version 1.2, Level 2 Revision 116.

[41] Trusted Computing Group. *TCG PC Client Specific Implementation Specification for Conventional BIOS*, February 2012. Revision 1.00, For TPM Family 1.2; Level 2.

[42] Trusted Computing Group. *Trusted Platform Module Library Part 1: Architecture*, September 2016. Family "2.0" Level 00 Revision 01.38.

[43] Rafal Wojtczuk and Corey Kallenberg. Attacks on UEFI security. in *CanSecWest*, Vancouver, British Columbia, March 2015.

[44] Liehuang Zhu, Zijian Zhang, Lejian Liao, and Cong Guo. A secure robust integrity reporting protocol of trusted computing for remote attestation under fully adaptive party corruptions. In Ying Zhang, editor, *Future Wireless Networks and Information Systems*, pages 211–217, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.