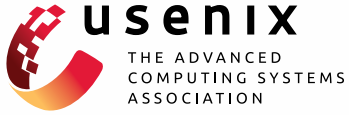


Ironclad Apps: End-to-End Security via Automated Full-System Verification

Chris Hawblitzel, Jon Howell, and Jacob R. Lorch, ***Microsoft Research;***
Arjun Narayan, ***University of Pennsylvania;*** **Bryan Parno,** ***Microsoft Research;***
Danfeng Zhang, ***Cornell University;*** **Brian Zill,** ***Microsoft Research***

https://www.usenix.org/conference/osdi14/technical-sessions/presentation/hawblitzel

This paper is included in the Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation.

October 6–8, 2014 • Broomfield, CO

978-1-931971-16-4

Open access to the Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

Ironclad Apps: End-to-End Security via Automated Full-System Verification

Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan[†], Bryan Parno, Danfeng Zhang[⋆], Brian Zill
Microsoft Research [†] University of Pennsylvania [⋆] Cornell University

Abstract

An Ironclad App lets a user securely transmit her data to a remote machine with the guarantee that every instruction executed on that machine adheres to a formal abstract specification of the app's behavior. This does more than eliminate implementation vulnerabilities such as buffer overflows, parsing errors, or data leaks; it tells the user exactly how the app will behave at all times. We provide these guarantees via complete, low-level software verification. We then use cryptography and secure hardware to enable secure channels from the verified software to remote users. To achieve such complete verification, we developed a set of new and modified tools, a collection of techniques and engineering disciplines, and a methodology focused on rapid development of verified systems software. We describe our methodology, formal results, and lessons we learned from building a full stack of verified software. That software includes a verified kernel; verified drivers; verified system and crypto libraries including SHA, HMAC, and RSA; and four Ironclad Apps.

1 Introduction

Today, when Alice submits her personal data to a remote service, she has little assurance that her data will remain secure. At best, she has vague legal guarantees provided by the service's privacy policy and the hope that the owner will follow industry best practices. Even then, a vulnerable OS, library, or application may undermine the service provider's best intentions [51].

In theory, complete formal verification of the service's code would replace this tenuous position with the strong mathematical guarantee that the service precisely matches Alice's formally specified security expectations. Unfortunately, while software verification provides strong guarantees [4, 6, 8, 17, 39], the cost is often high [25, 35, 36]; e.g., seL4 took over 22 person-years of effort to verify a microkernel. Some strong guarantees have been obtained in much less time, but those guarantees depend on unverified lower-level code. For example, past work produced a verified TLS implementation [9] and a proof of correctness for RSA-OAEP [7]. In both cases, though, they assumed the crypto libraries, their runtimes (e.g., .NET), and the OS were correct.

In contrast, we aim to create *Ironclad Apps* that are verifiably end-to-end secure, meaning that: (1) The verification covers *all* code that executes on the server, not just the app but also the OS, libraries, and drivers. Thus, it does not assume that any piece of server software is correct. (2) The proof covers the *assembly code* that gets executed, not the high-level language in which the app is written. Thus, it assumes that the hardware is correct, but assumes nothing about the correctness of the compiler or runtime. (3) The verification demonstrates *remote equivalence*: that to a remote party the app's implementation is *indistinguishable* from the app's high-level abstract state machine.

Verifiable remote equivalence dictates the behavior of the entire system in every possible situation. Thus, this approach provides stronger guarantees than type checkers or tools that look for classes of bugs such as buffer overflows or bounds errors. Our proof of remote equivalence involves proving properties of both functional correctness and information flow; we do the latter by proving *noninterference*, a relationship between two runs of the same code with different inputs.

We then show how remote equivalence can be strengthened to *secure* remote equivalence via Trusted Computing [3, 53]. Specifically, the app verifiably uses secure hardware, including a TPM [63], to convince a remote client that its public key corresponds to a private key known only to the app. The client uses the public key to establish a secure channel, thereby achieving security equivalent to direct communication with the abstractly specified app [30].

Another goal of our work is to make it feasible to build Ironclad Apps with modest developer effort. Previous efforts, such as seL4 [35] or VCC [13], took tens of person-years to verify one software layer, so verifying an entire stack using these techniques may be prohibitive. To reduce developer effort, we use state-of-the-art tools for *automated* software verification, such as Dafny [39], Boogie [4], and Z3 [17]. These tools need much less guidance from developers than interactive proof assistants used in previous work [35, 52].

However, many in the verification community worry that automated verification cannot scale to large software and that the tools' heuristics inevitably lead to unstable verification results. Indeed, we encountered these challenges, and dealt with them in multiple ways: via two new tools (§3.4); via modifications to existing verification tools to support *incremental verification*, *opaque functions*, and *automatic requirement propagation*; via software engineering disciplines like *premium functions* and *idiomatic specification*; via a *nonlinear math library* that

lets us suppress instability-inducing arithmetic heuristics; and via *provably correct libraries* for performing crypto operations and manipulating arrays of bits, bytes, and words. All these contributions support stable, automated, large-scale, end-to-end verification of systems software.

To demonstrate the feasibility of our approach, we built four Ironclad Apps, each useful as a standalone service but nevertheless compactly specifiable. For instance, our Notary app securely assigns logical timestamps to documents so they can be conclusively ordered. Our other three apps are a password hasher, a multi-user trusted counter [40], and a differentially-private database [19].

We wrote nearly all of the code from scratch, including the apps, libraries, and drivers. For the OS, we used the Verve microkernel [65], modified to support secure hardware and the Dafny language. For our four apps collectively we wrote about 6K lines of implementation and 30K lines of proof annotations. Simple benchmarks experience negligible slowdown, but unoptimized asymmetric crypto workloads slow down up to two orders of magnitude.

Since we prove that our apps conform to their specifications, we want these specs to be small. Currently, the total spec size for all our apps is 3,546 lines, satisfying our goal of a small trusted computing base (TCB).

2 Goals and Assumptions

Here we summarize Ironclad's goals, non-goals, and threat model. As a running example, we use our Notary app, which implements an abstract Notary state machine. This machine's state is an asymmetric key pair and a monotonic counter, and it signs statements assigning counter values to hashes. The crypto lets a user securely communicate with it even over an untrusted network.

2.1 Goals

Remote equivalence. Any remote party, communicating with the Ironclad App over an untrusted network, should receive the same sequence of messages as she would have received if she were communicating with the app's abstract state machine over an untrusted network. For example, the Notary app will never roll back its counter, leak its private key, sign anything other than notarizations, compute signatures incorrectly, or be susceptible to buffer overflows, integer overflows, or any other implementation-level vulnerabilities.

Secure channel. A remote user can establish a secure channel to the app. Since this protects the user's communication from the untrusted network, the remote equivalence guarantee leads to security commensurate with actual equivalence. For example, the Notary's spec says it computes its key pair using secure randomness, then obtains an attestation binding the public key and the app's code to a secure platform. This attestation convinces a remote user that a notarization signed with the corresponding private key was generated by the Notary's code, which is equivalent to the abstract Notary state machine. Note that not all messages need to use the secure channel; e.g., hashes sent to the Notary are not confidential, so the app does not expect them to be encrypted.

Completeness. Every software component must be either verified secure or run in a verified-secure sandbox; our current system always uses the former option. The assurance should cover the entire system as a coherent whole, so security cannot be undermined by incorrect assumptions about how components interact. Such gaps introduced bugs in previous verification efforts [65].

Low-level verification. Since complex tools like compilers may introduce bugs (a recent study found 325 defects in 11 C compilers [66]), we aim to verify the actual instructions that will execute rather than high-level code. Verifying assembly also has a potential performance benefit: We can hand-tune our assembly code without fear of introducing bugs that violate our guarantees.

Rapid development by systems programmers. To push verification towards commercial practicality, we need to improve the scale and functionality of verification tools to support large, real-world programs. This means that non-expert developers should be able to rapidly write and efficiently maintain verified code.

2.2 Non-goals

Compatibility. Ideally, we would verify existing code written in standard languages. However given the challenges previous efforts have faced [13], we choose to focus on fresh code written in a language designed to support verification. If we cannot achieve the goals above in such a setting, then we certainly cannot achieve it in the challenging legacy setting.

Performance. Our primary goal is to demonstrate the feasibility of verifying an entire software stack. Hence, we focus on single-core machines, poll for network packets rather than using interrupts, and choose algorithms that facilitate proofs of correctness rather than performance.

However, verification gives us a strong safety net with which to perform arbitrarily aggressive optimizations, since we can count on our tools to catch any errors that might be introduced. We exploited this repeatedly.

2.3 Threat model and assumptions

Ironclad provides security against software-based attackers, who may run arbitrary software on the machine before the Ironclad App executes and after it terminates. The adversary may compromise the platform's firmware, BIOS, and peripheral devices, such as the network card. We assume the CPU, memory, and chipset are correct, and the attacker does not mount physical attacks, such as electrically probing the memory bus.

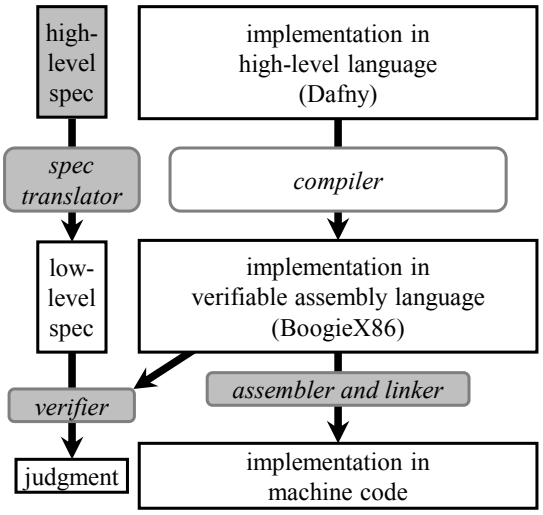


Figure 1: **Methodology Overview.** *Rounded rectangles represent tools; regular rectangles represent artifacts. Trusted components are shaded.*

We focus on privacy and integrity; we do not prove liveness, so attacks or bugs may result in denial of service. Our hardware model is currently inadequate to prove the absence of side channels due to cache or timing effects.

We assume the platform has secure hardware support, specifically a *Trusted Platform Module* (TPM). Deployed on over 500 million computers [64], the TPM provides a hardware-based root of trust [3, 53, 63]. That is, it records information about all software executed on the platform during a boot cycle in a way that can be securely reported, via an *attestation* protocol, to a remote party. The TPM maintains records about the current boot cycle in the form of hash chains maintained in volatile *Platform Configuration Registers* (PCRs). Software can add information to the PCRs via an *extend* operation. This operation updates a PCR to the hash of its previous value concatenated with the new information. The TPM also has a private RSA key that never leaves the device and can be used to *attest* to the platform's current state by signing the PCR values. The TPM's manufacturer certifies that the corresponding public key is held by a real hardware TPM, preventing impersonation by software. Finally, the TPM provides access to a stream of secure random bytes.

3 The Ironclad Methodology

This section describes our methodology for verifying Ironclad Apps are secure and for efficiently building them.

3.1 Overview

Previous verification efforts required >20 person-years of effort to develop relatively small verified software. Since we aim to perform low-level, full-system verification with modest effort, our methodology (Fig. 1) differs from previous efforts in significant ways.

With Ironclad, we use a verification stack based on Floyd-Hoare reasoning (§3.2) to prove the functional correctness of our code. We write both our specifications (§3.3) and code (§3.4) in Dafny [39], a remarkably usable high-level language designed to facilitate verification. Unlike tools used in previous efforts, Dafny supports automated verification via the Z3 [17] SMT solver, so the tool often automatically fills in low-level proof details.

Given correct Dafny code, we built automated tools to translate our code to BoogieX86 [65], a verifiable assembly language (§3.4). The entire system is verified at the assembly level using the Boogie verifier [4], so any bugs in Dafny or in the compiler will be caught at this stage.

At every stage, we use and extend existing tools and build new ones to support rapid development of verified code (§3.5), using techniques like real-time feedback in developer UIs and multi-level verification result caching.

Finally, since many security properties cannot be expressed via functional correctness, we develop techniques for verifying *relational properties* of our code (§3.6).

If all verification checks pass, a simple trusted assembler and linker produces the machine code that actually runs. We run that code using the platform's secure late-launch feature (§6.1), which puts the platform into a known-good state, records a hash of the code in the TPM (§2.3), then starts executing verified code. These steps allow remote parties to verify that Ironclad code was indeed properly loaded, and they prevent any code that runs before Ironclad, including the boot loader, from interfering with its execution.

3.2 Background: Floyd-Hoare verification

We verify Ironclad Apps using Floyd-Hoare reasoning [21, 31]. In this approach, programs are annotated with assertions about program state, and the verification process proves that the assertions will be valid when the program is run, for all possible inputs. As a simple example, the following program is annotated with an assertion about the program state at the end of a method (a "post-condition"), saying that the method output `O` must be an even number:

```
method Main(S, I) returns(O)
    ensures even(O);
{ O := (S + S) + (I + I); }
```

A tool like Dafny or Boogie can easily and automatically verify that the postcondition above holds for all possible inputs `S` and `I`.

For a long-running program with multiple outputs, we can specify a restriction on *all* of the program's outputs by annotating its output method with a precondition. For instance, writing:

```
method WriteOutput(O)   // Trusted output
    requires even(O);   // method
```

ensures that the verifier will reject code unless, like the following, it can be proven to only output even numbers:

```
method Main() {
  var count := 0;
  while(true) invariant even(count) {
    count := count + 2;
    WriteOutput(count);
  } }
```

Boogie and Dafny are sound, i.e., they will never approve an incorrect program, so they cannot be complete, i.e., they will sometimes fail to automatically recognize valid programs as correct. Thus, they typically require many preconditions, postconditions, and loop invariants inside the program to help them complete the verification, in addition to the preconditions and postconditions used to write the trusted specifications. The loop invariant `invariant even(count)` in the example above illustrates this: it is not part of the trusted specification, but instead serves as a hint to the verification tool.

By itself, Floyd-Hoare reasoning proves safety properties but not liveness properties. For example, a postcondition establishes a property of the state upon method exit, but the method may fail to terminate. We have not proven liveness for Ironclad Apps.

3.3 Writing trustworthy specifications

To build Ironclad Apps, we write two main types of specifications: hardware and apps. For hardware specs, since we aim for low-level verification, we write a specification for each of the ∼56 assembly instructions our implementation will use. An instruction's spec describes its preconditions and its effects on the system. For example, `Add` ensures that the sum of the input registers is written to the destination register, and requires that the input values not cause the sum to overflow.

For app specs, we write abstract descriptions of desired app behavior. These are written modularly in terms of lower-level library specs. For example, the spec for the Notary describes how the app's state machine advances and the outputs permitted in each state; one possible output is a signed message which is defined in terms of our spec for RSA signing.

The verification process removes all implementation code from the TCB by proving that it meets its high-level spec given the low-level machine spec. However, the specs themselves *are* part of the TCB, so it is crucial that they be worthy of users' trust. To this end, we use *spec-first design*, *idiomatic specification*, and *spec reviews*.

Spec-first design. To encourage spec quality, we write each specification before starting on its implementation. This order makes the spec likely to express desired properties rather than a particular mechanism. Writing the spec afterwards might port implementation bugs to the spec.

Idiomatic specification. To ensure trustworthy specs, we aim to keep them small and simple, making bugs less likely and easier to spot. We accomplish this by specifying only the feature subset that our system needs, and by ensuring that the implementation cannot trigger other features; e.g., our verifier will not permit any assembly instructions not in the hardware spec. This is crucial for devices; e.g., the TPM's documentation runs to hundreds of pages, but we need only a fraction of its functionality. Hence, our TPM spec is only 296 source lines of code (SLOC).

Spec reviews. We had two or more team members develop each spec, and another review their work independently. This caught several bugs before writing any code.

Despite our techniques, specs may still contain bugs. However, we expect them to contain significantly fewer bugs than implementations. First, our specs are smaller (§8.1). Second, our specs are written in a more abstract, declarative fashion than implementation code, making spec bugs both less likely to occur and easier to find when they do occur. For example, one line in our Notary spec (§5.1) says that a number representing a counter is incremented. The code *implementing* that addition, in contrast, involves hundreds of lines of code: it implements the unbounded-precision number using an array of machine words, so addition must handle carries and overflow.

Overall, our experience (§7) suggests specs are indeed more trustworthy than code.

3.4 Producing verifiable assembly language

To enable rapid, large-scale software development while still verifying code at a low level, we take a two-layer verification approach (Figure 1): we write our specs and implementation in the high-level Dafny language, but we re-verify the code after compiling to assembly language.

We replaced the existing Dafny compiler targeting .NET and Windows with two new components, a trusted spec translator and a new untrusted compiler called DafnyCC. The trusted spec translator converts a tiny subset of Dafny into BoogieX86. This subset includes just those features useful in writing specs: e.g., functions, type definitions, and sequences, but not arrays.

Our untrusted DafnyCC compiler, in contrast, consumes a large subset of the Dafny language. It translates both the code *and* the proofs written in Dafny into BoogieX86 assembly that Boogie can automatically verify. It also automatically inserts low-level proofs that the stack is used safely (§6.3), that OS invariants are maintained (§6.4), etc. Because all of the code emitted by DafnyCC is verified by Boogie, none of its complexity is trusted. Thus, we can add arbitrarily complex features and optimizations without hurting security. Indeed, Boogie caught several bugs made during compilation (§7.7).

3.5 Rapid verification

A key goal of Ironclad is to reduce the verification burden for developers, so we use the following techniques to support rapid verification.

Preliminary verification. Although ultimately we must verify code at the assembly level, it is useful to perform a fast, preliminary verification at the Dafny level. This lets the developer quickly discover bugs and missing proof annotations. The verification is particularly rapid because Dafny includes a plugin for the Visual Studio interactive development environment that verifies code incrementally as the developer types, emitting error messages and marking the offending code with squiggly underlines.

Modular verification. We added support to Dafny for modular verification, allowing one file to import another file's interfaces without reverifying that code.

Shared verification. Our *IronBuild* tool shares verification results among developers via a cloud store. Since each developer verifies code before checking it in, whenever another developer checks out code, verification will succeed immediately based on cached results. IronBuild precisely tracks dependencies by hash to ensure fidelity.

3.6 Verifying relational properties

For Ironclad Apps, we prove properties beyond functional correctness, e.g., that the apps do not leak secrets such as keys. Although standard Floyd-Hoare tools like Boogie and Dafny focus on functional correctness, we observed that we could repurpose a Boogie-based experimental tool, SymDiff [37], to prove *noninterference* properties. We combine these proofs with our functional correctness proofs to reason about the system's security (§4).

Suppose that variable `S` represents a secret inside the program and `I` represents a public input to the program. The statement $\texttt{O} := (\texttt{S}+\texttt{S})+(\texttt{I}+\texttt{I})$ satisfies a functional correctness specification `even(O)`. However, in doing so, it leaks information about the secret `S`.

The statement $\texttt{O} := (\texttt{S}-\texttt{S})+(\texttt{I}+\texttt{I})$, by contrast, satisfies `even(O)` yet leaks no information about `S`. Intuitively, the value stored in `O` depends on `I` but is independent of `S`. The concept of noninterference [24, 57, 61] formalizes this intuition by reasoning about multiple executions of a program, and comparing the outputs to see which values they depend on. Suppose that we pass the same public input `I` to all the executions, but vary the secret `S` between the executions. If all the executions produce the same output `O` regardless of `S`, then `O` is independent of `S`, and the program leaks nothing about `S`.

Mathematically, noninterference means that for all possible pairs of executions, if the public inputs `I` are equal but the secrets `S` may be different, then the outputs `O` are equal. (Some definitions also require that termination is independent of secrets [57], while others do not [61]; for simplicity, we use the latter.) More formally, if we call the two executions in each pair L and R, for left and right, then noninterference means $\forall \texttt{S}_L, \texttt{S}_R \,.\, \texttt{I}_L = \texttt{I}_R \implies \texttt{O}_L = \texttt{O}_R$. For instance, $\texttt{O} := (\texttt{S}-\texttt{S})+(\texttt{I}+\texttt{I})$ satisfies this condition, but $\texttt{O} := (\texttt{S}+\texttt{S})+(\texttt{I}+\texttt{I})$ does not.

To allow the SymDiff tool to check noninterference, we annotate some of our code with explicit relational annotations [5], writing x_L as `left(`x`)` and x_R as `right(`x`)`:

```
method Test(S, I) returns(O)
    requires left(I) == right(I);
    ensures left(O) == right(O);
    ensures even(O);
{ O := (S - S) + (I + I); }
```

The relational precondition `left(I) == right(I)` means SymDiff must check that $\texttt{I}_L = \texttt{I}_R$ wherever `Test` is called, and the relational postcondition `left(O) == right(O)` means SymDiff must check that this method ensures $\texttt{I}_L = \texttt{I}_R \implies \texttt{O}_L = \texttt{O}_R$.

However, for most of our code, SymDiff leverages our existing functional correctness annotations and does not need relational annotations. For example, SymDiff needs only the functional postcondition in this code:

```
method ComputeIpChecksum(I) returns(O)
    ensures O == IpChecksum(I);
```

to infer that if $\texttt{I}_L = \texttt{I}_R$, then $\texttt{IpChecksum}(\texttt{I}_L) = \texttt{IpChecksum}(\texttt{I}_R)$, so $\texttt{O}_L = \texttt{O}_R$.

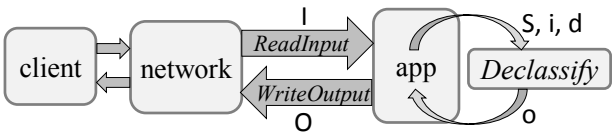
4 Proving Ironclad Security Properties

This section describes how we combine the previous section's ideas of functional correctness, like `even(O)`, and noninterference, like $\texttt{I}_L = \texttt{I}_R \implies \texttt{O}_L = \texttt{O}_R$, to prove the security of our Ironclad Apps. It describes the architecture, theorems, and proofs at a high level. In §5, we show how they are instantiated for each app, and in §6 we give details about the key lemmas we prove about our system to support these high-level results.

4.1 Declassification and the Ironclad architecture

Pure noninterference establishes that a program's output values are completely independent of the program's secrets, but this requirement is too strong for most real-world systems. In practice, programs deliberately allow limited influence of the secrets on the output, such as using a secret key to sign an output. A security policy for such programs explicitly *declassifies* certain values, like a signature, so they can be output despite being dependent on secrets.

Figure 2 shows the overall structure of the Ironclad system, including an abstract declassifier that authorizes the release of selected outputs derived from secrets. We express each app's declassification policy as a state machine, thereby binding the release of secret-derived data to the high-level behavior of the abstract app specification. We assume that the client communicates with the Ironclad App across a network that may drop, delay, duplicate, or mangle data. The network, however, does not have access to the app's secrets. The app receives some possibly-mangled inputs `I` and responds by sending some outputs `O` to the network, which may mangle `O` before passing them to the client. While computing the outputs `O`, the app may




```
method ReadInput() returns(I);
  ensures left(I) == right(I);

method Declassify(S, i, d) returns(o);
  requires d == StateMachineOutput(S, i);
  requires left(i) == right(i);
  ensures left(o) == right(o);

method WriteOutput(O);
  requires left(O) == right(O);
```

Figure 2: **Abstract system structure and trusted input/output/declassify specification.**

appeal to the declassification policy as many times as it wishes. Each time, it passes its secrets S, some inputs i, and the desired declassified outputs d to the declassifier. For verification to succeed, the desired outputs must equal the outputs according to the abstract state machine's policy: $\mathtt{d} = \mathit{StateMachineOutput}(\mathtt{S},\mathtt{i})$. If static verification proves that the declassification policy is satisfied, the declassifier produces declassified outputs o that the app can use as part of its outputs O.

In the real implementation, o simply equals d, so that the declassifier is a no-op at run-time. Nevertheless, we hide this from the verifier, because we want to reveal $\mathtt{o}_L = \mathtt{o}_R$ without revealing $\mathtt{d}_L = \mathtt{d}_R$; in some cases where the secrets S are in principle computable by brute-force search on d (e.g., by factoring an RSA public key), $\mathtt{d}_L = \mathtt{d}_R$ might imply $\mathtt{S}_L = \mathtt{S}_R$, which we do not want.

4.2 Ironclad security theorems

Given the execution model described above, for each of our apps, we first prove functional correctness as a precondition for declassification:

Theorem 1 FUNCTIONAL CORRECTNESS. *At each declassification Declassify*(S,i,d)*, the desired outputs d satisfy the app's functional correctness policy, according to the app's abstract state machine:* $d = \mathit{StateMachineOutput}(S,i)$.

In other words, we only declassify values that the abstract state machine would have output; the state machine clearly considers these values safe to output.

Second, we split noninterference into two parts and prove both: noninterference along the path from the inputs I to the declassifier, and noninterference along the path from the declassifier to the outputs O:

Theorem 2 INPUT NONINTERFERENCE. *At each declassification Declassify*(S,i,d), $I_L = I_R \Longrightarrow i_L = i_R$.

In other words, the declassifier's public inputs i may depend on inputs from the network I, but not on secrets S.

Theorem 3 OUTPUT NONINTERFERENCE. *Each time the program outputs* O, $I_L = I_R \wedge o_L = o_R \Longrightarrow O_L = O_R$.

In other words, the outputs O may depend on inputs from the network I and on any declassified values o, but not on secrets S.

As discussed in more detail in later sections, we carried out formal, mechanized proofs of these three theorems using the Boogie and SymDiff tools for each Ironclad App.

These theorems imply remote equivalence:

Corollary 1 REMOTE EQUIVALENCE. *To a remote party, the outputs received directly from the Ironclad App are equal to the outputs generated by the specified abstract state machine over some untrusted network, where the state machine has access to the trusted platform's secrets, but the untrusted network does not. (Specifically, if the Ironclad App generates some outputs* O_L *and the untrusted network generates some outputs* O_R*, then* $O_L = O_R$*.)*

Proof Sketch: We prove this by constructing an alternative, abstract counterpart to the Ironclad App. Label the real Ironclad App *L* and the counterpart *R*. The counterpart *R* consists of two components: the specified abstract state machine, which can read the trusted platform's secrets S, and an untrusted network, which cannot. We construct *R* by using the actual Ironclad App code as the untrusted network, with two changes. First, in the untrusted network, we replace the real secrets S with an arbitrary value S′, modeling the network's lack of access to the real secrets S. Second, we replace *R*'s ordinary declassifier with the abstract state machine producing $\mathtt{o}_R = \mathit{StateMachineOutput}(\mathtt{S},\mathtt{i}_R)$, modeling the abstract state machine's access to the real secrets S. We pass the same input to both the real Ironclad App *L* and to *R*, so that $\mathtt{I}_L = \mathtt{I}_R$. By INPUT NONINTERFERENCE, the inputs to the declassifier are the same: $\mathtt{i}_L = \mathtt{i}_R$. The real declassifier simply returns $\mathtt{o}_L = \mathtt{d}_L$, and by FUNCTIONAL CORRECTNESS, the real Ironclad App produces the outputs $\mathtt{d}_L = \mathit{StateMachineOutput}(\mathtt{S},\mathtt{i}_L)$. Since $\mathtt{i}_L = \mathtt{i}_R$ and we pass the same secrets S to both *L* and *R*, we conclude that $\mathtt{o}_L = \mathtt{d}_L = \mathit{StateMachineOutput}(\mathtt{S},\mathtt{i}_L) = \mathit{StateMachineOutput}(\mathtt{S},\mathtt{i}_R) = \mathtt{o}_R$. Then by OUTPUT NONINTERFERENCE, both *L* and *R* generate the same outputs: $\mathtt{O}_L = \mathtt{O}_R$. ■

This shows that the Ironclad App's output is the same as that of the abstract state machine and an untrusted network. Thus, the output a remote party sees, which is produced by the Ironclad App and an *actual* untrusted network, is the same as that of the abstract state machine and an untrusted network composed of the actual and chosen untrusted networks.

4.3 Limitations of this model

Since we have not formally proven liveness properties like termination, an observer could in principle learn informa-

```
datatype NotaryState = NotaryState_c(
  keys:RSAKeyPair, cntr:nat);

predicate NotarizeOpCorrect(
  in_st:NotaryState, out_st:NotaryState,
  in_msg:seq<int>, out_stmt:seq<int>,
  out_sig:seq<int>)
{
 ByteSeq(in_msg)
 && out_st.keys == in_st.keys
 && out_st.cntr == in_st.cntr + 1
 && out_stmt==[OP_COUNTER_ADV]
    + rfc4251_encode(out_st.cntr) + in_msg
 && out_sig==RSASign(in_st.keys, out_stmt)
}
```

Figure 3: **Part of the Notary Spec.** *Simplified for brevity and clarity, this is a predicate the implementation must satisfy before being allowed to declassify* `out_sig`*, which otherwise cannot be output because it depends on secret data.*

tion about the secrets from whether an output was generated for a given input. Also, we have not formally proved timing properties, so an observer could also learn information from a timing channel. To eliminate the possibility of such timing-based information leakages, in the future we would like to prove that the time of the outputs is independent of secrets. The literature contains many possible approaches [10, 15, 26]; for example, we might prove an upper bound on the time taken to produce an output, and delay each output until the upper bound is reached.

5 Ironclad Applications

To make the guarantees of remote equivalence concrete, we describe the four apps we built. The proof for each app, in turn, builds on lemmas about lower-level libraries, drivers, and OS, which we discuss in §6.

Each app compiles to a standalone system image that communicates with other machines via UDP. Nevertheless, each is a useful complete application that would merit at least one dedicated machine in a data center. In the future, hardware support for fine-grained secure execution environments [42] may offer a simple path towards multiplexing Ironclad Apps.

5.1 Notary

Our Notary app securely assigns logical timestamps to documents so they can be conclusively ordered. This is useful, e.g., for establishing patent priority [28] or conducting online auctions [62]. Typically, users of such a service must trust that some machine is executing correct software, or that at least k of n machines are [12]. Our Ironclad Notary app requires no such assumption.

Lemma 1 NOTARY REMOTE EQUIVALENCE. *The Notary app is remotely equivalent to a state machine with the following state:*

- $\langle \mathsf{PublicKey}, \mathsf{PrivateKey} \rangle$*, computed using the RSA key generation algorithm from the first consecutive sequence of random bytes read from the TPM;*
- *a TPM, whose PCR 19 has been extended with the public part of that key pair; and*
- *a* Counter*, initialized to 0;*

and the following transitions:

- *Given input* $\langle \texttt{connect}, \mathsf{Nonce} \rangle$*, it changes the TPM state by obtaining a quote* Quote *over PCRs 17–19 and external nonce* Nonce*. It then outputs* $\langle \mathsf{PublicKey}, \mathsf{Quote} \rangle$.
- *Given input* $\langle \texttt{notarize}, \mathsf{Hash} \rangle$*, it increments* Counter *and returns* $\mathsf{Sig}_{\mathsf{PrivateKey}}(\texttt{OP-CTR-ADV} \parallel \mathsf{RFC4251Encode}(\mathsf{Counter}) \parallel \mathsf{Hash})$.

Figure 3 shows part of the corresponding Dafny spec.

Proving this lemma required proofs of the following. (1) Input non-interference: the nonce and message the app passes the declassifier are based solely on public data. (2) Functional correctness of `connect`: the app derives the key from randomness correctly, and the TPM quote the app obtains comes from the TPM when its PCRs are in the required state. (3) Functional correctness of `notarize`: the app increments the counter and computes the signature correctly. (4) Output non-interference: Writes to unprotected memory depend only on public data and the computed state machine outputs.

Proving remote-equivalence lemmas for the other apps, which we describe next, required a similar approach.

5.2 TrInc

Our trusted incrementer app, based on TrInc [40], generalizes Notary. It maintains per-user counters, so each user can ensure there are no gaps between consecutive values. It is a versatile tool in distributed systems, useful e.g. for tamper-resistant audit logs, Byzantine-fault-tolerant replicated state machines, and verifying that an untrusted file server behaves correctly.

Lemma 2 TRINC REMOTE EQUIVALENCE. *The TrInc app is remotely equivalent to a state machine like Notary's except that it has multiple counters, each a tuple* $\langle K_i, v_i \rangle$*, and a meta-counter initially set to 0. In place of the* `notarize` *transition it has:*

- *Given input* $\langle \texttt{create}, K \rangle$*, it sets* $i := meta\_counter$*, increments* $meta\_counter$*, and sets* $\langle K_i, v_i \rangle = \langle K, 0 \rangle$.
- *Given input* $\langle \texttt{advance}, i, v_{new}, \mathsf{Msg}, \mathsf{UserSig} \rangle$*, let* $v_{old} = v_i$ *in counter tuple* i*. If* $v_{old} \leq v_{new}$ *and* $\mathsf{VerifySig}_{K_i}(v_{new} \parallel \mathsf{Msg}, \mathsf{UserSig})$ *succeeds, it sets* $v_i := v_{new}$ *and outputs* $\mathsf{Sig}_{\mathsf{PrivateKey}}(\texttt{OP-CTR-ADV} \parallel \mathsf{encode}(i) \parallel \mathsf{encode}(v_{old}) \parallel \mathsf{encode}(v_{new}) \parallel \mathsf{Msg})$.

5.3 Password hasher

Our next app is a password-hashing appliance that renders harmless the loss of a password database. Today, attackers frequently steal such databases and mount offline attacks.

Even when a database is properly hashed and salted, low-entropy passwords make it vulnerable: one study recovered 47–79% of passwords from low-value services, and 44% of passwords from a high-value service [41].

Lemma 3 PASSHASH REMOTE EQUIVALENCE. *The PassHash app is remotely equivalent to the following state machine. Its state consists of a byte string* Secret, *initialized to the first 32 random bytes read from the TPM. Given input* ⟨`hash`, Salt, Password⟩, *it outputs* SHA256(Secret ‖ Salt ‖ Password).

Meeting this spec ensures the hashes are useless to an offline attacker: Without the secret, a brute-force guessing attack on even the low-entropy passwords is infeasible.

5.4 Differential-privacy service

As an example of a larger app with a more abstract spec, we built an app that collects sensitive data from contributors and allows analysts to study the aggregate database. It guarantees each contributor *differential privacy* [19]: the answers provided to the analyst are virtually indistinguishable from those that would have been provided if the contributor's data were omitted. Machine-checked proofs are especially valuable here; prior work [46] showed that implementations are prone to devastating flaws.

Our app satisfies Dwork's formal definition: An algorithm $\mathcal{A}$ is differentially private with privacy ε if, for any set of answers $\mathcal{S}$ and any pair of databases D_1 and D_2 that differ by a single row, $P[\mathcal{A}(D_1) \in \mathcal{S}] \leq \lambda \cdot P[\mathcal{A}(D_2) \in \mathcal{S}]$, where we use the privacy parameter $\lambda = e^{\varepsilon}$ [23].

Privacy budget. Multiple queries with small privacy parameters are equivalent to a single query with the product of the parameters. Hence we use a *privacy budget* [20]. Beginning with the budget $b = \lambda$ guaranteed to contributors, each query Q with parameter λ_Q divides the budget $b' := b/\lambda_Q$; a query with $\lambda_Q > b$ is rejected.

Noise computation. We follow the model of Dwork et al. [20]. We first calculate Δ, the *sensitivity* of the query, as the most the query result can change if a single database row changes. The analyst receives the sum of the true answer and a random noise value drawn from a distribution parameterized by Δ.

Dwork et al.'s original algorithm uses noise from a Laplace distribution [20]. Computing this distribution involves computing a natural logarithm, so it cannot be done precisely on real hardware. Thus, practical implementations simulate this real-valued distribution with approximate floating point values. Unfortunately, Mironov [46] devised a devastating attack that exploits information revealed by error in low-order bits to reveal the *entire database*, and showed that all five of the main differential-privacy implementations were vulnerable.

To avoid this gap between proof and implementation, we instead use a noise distribution that only involves rational numbers, and thus can be sampled precisely using the x86 instruction set. In our specification, we model these rational numbers with real-valued variables, making the spec clearer and more compact. We then prove that our 32-bit-integer-based implementation meets this spec.

```
predicate DBsSimilar(d1:seq<Row>,d2:seq<Row>)
  |d1| == |d2| &&
  exists diff_row ::
    forall i :: 0 <= i < |d1| && i != diff_row
     ==> d1[i] == d2[i]

predicate SensitivitySatisfied(prog:seq<Op>,
                min:int, max:int, delta:int)
  forall d1:seq<Row>, d2:seq<Row> ::
  Valid(d1)&&Valid(d2)&&DBsSimilar(d1, d2)==>
  -delta <= MapperSum(d1, prog, min, max) -
            MapperSum(d2, prog, min, max)
         <= delta
```

Figure 4: **Summing Reducer Sensitivity.** *Our differential-privacy app is verified to satisfy a predicate like this, relating reducer output sensitivity to the* Δ *used in noise generation.*

Lemma 4 DIFFPRIV REMOTE EQUIVALENCE. *The DiffPriv app is remotely equivalent to a state machine with the following state:*

- *key pair and TPM initialized as in Notary;*
- *remaining budget b, a real number; and*
- *a sequence of rows, each consisting of a duplicate-detection nonce and a list of integer column values;*

and with transitions that connect to the app, initialize the database, add a row, and perform a query.

We also prove a higher-level property about this app:

Lemma 5 SENSITIVITY. *The value* Δ *used as the sensitivity parameter in the spec's noise computation formula is the actual sensitivity of the query result. That is, if we define* $\mathcal{A}(D)$ *as the answer the app computes when the database is D, then for any two databases* D_1 *and* D_2, $|\mathcal{A}(D_1) - \mathcal{A}(D_2)| \leq \Delta$.

To make this verifiable, we use Airavat-style queries [56]. That is, each query is a *mapper*, which transforms a row into a single value, and a *reducer*, which aggregates the resulting set; only the latter affects sensitivity. The analyst can provide an arbitrary mapper; we provide, and prove sensitivity properties for, the single reducer `sum`. It takes RowMin and RowMax parameters, clipping each mapper output value to this range. Figure 4 shows the property we verified: that the sensitivity of `sum` is $\Delta = \text{RowMax} - \text{RowMin}$ regardless of its mapper-provided inputs.

6 Full-System Verification

We have mechanically verified the high-level theorems described in §4. Although the mechanical verification uses automated theorem proving, the code must contain manual annotations, such as loop invariants, preconditions, and postconditions (§3.2). One can think of these

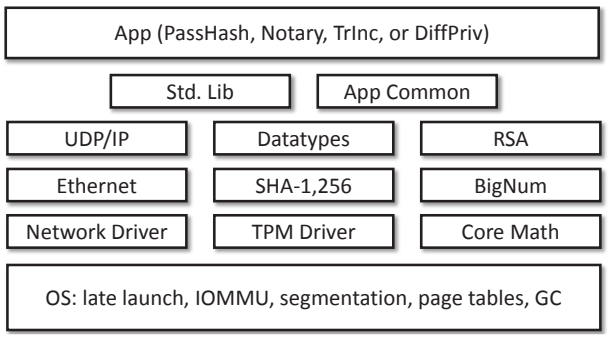


Figure 5: **System Overview.**

annotations, spread throughout the code, as lemmas that build to the final high-level theorems.

To convey the work necessary to complete the verification, this section gives a sampling of the key lemmas we proved along the way. For clarity and conciseness, we state each lemma as brief English text; the real mechanical "lemmas" are the annotations in the code itself. The lemmas described in this section are not, on their own, sufficient for the proof, since they are only a sampling. Nevertheless, a failure in any of the lemmas below would cause the high-level theorems to fail; we would not be able to establish the overall correctness of an Ironclad App if, for example, the cryptographic library or the garbage collector were incorrect.

6.1 Memory, devices, and information flow

Lemma 6 IOMMU CONFIGURATION. *The Ironclad Apps configure the IOMMU to divide memory into device-accessible and app-private memory; non-device operations access only app-private memory.*

Our assembly language instruction specifications check that non-device memory operations only access app-private memory that has been protected by the hardware's device exclusion vector, a simple IOMMU.

Commodity CPUs from AMD [1] and Intel [32] provide a *dynamic root-of-trust for measurement* (DRTM) feature, a.k.a. *late launch* [53]. It resets the CPU to a known state, stores a measurement (hash) of the in-memory code pointed to by the instruction's argument, and jumps to that code. After a late launch, the hardware provides the program control of the CPU and 64 KiB of protected memory. To use more than 64 KiB, it must first extend the IOMMU's protections, using our specification for IOMMU configuration. Only then can the program satisfy the preconditions for assembly language instructions accessing memory outside the 64-KiB region.

Lemma 7 DEVICES SEE NO SECRETS. *Only non-secret data is passed to devices.*

Our assembly language instruction specifications require that stores to device-accessible memory, i.e., memory that the IOMMU allows devices to see, can only store non-secret data $\circ$. In §3.6's terminology, non-secret means that $\circ_L = \circ_R$. More specifically, we require that the left and right executions generate the same sequence of device stores: the same values to the same addresses, modulo timing and liveness.

To prove $\circ_L = \circ_R$, we annotate our implementation's input and output paths with relational annotations. These input and output paths include the application event loops and the networking stack. For example, the Ethernet, IP, and UDP layers maintain relational properties on packets.

Lemma 8 KEY IN TPM. *Apps correctly extend a public key into the TPM's PCR 19. The private key is generated using TPM randomness and never leaves the platform.*

Lemma 9 ATTESTATION. *Apps generate a correct TPM attestation after extending their public key into a PCR.*

Corollary 2 SECURE CHANNEL. *If a remote client receives a public key and an attestation, and the attested PCR code values (PCRs 17, 18) match those of an Ironclad App, and the attested PCR data values (PCR 19) match the public key, and a certificate shows the attestation is from a legitimate hardware TPM manufacturer, then the client can use the public key to establish a secure channel directly to the Ironclad App.*

6.2 Cryptographic libraries

Lemma 10 HASHING. *Our SHA-{1,256} conforms to FIPS 180-4 [50], and our HMAC to FIPS 198-1 [49].*

Lemma 11 RSA OPERATIONS. *RSA keys are generated using consecutive randomness from the TPM (not selectively sampled), and pass the Miller-Rabin primeness test [45, 54]. Our implementations of RSA encrypt, decrypt, sign, and verify, including padding, produce byte arrays that conform to PKCS 1.5 and RSA standards [33].*

For basic cryptographic primitives such as hash functions, functional correctness is the best we can hope to verify. For instance, there is no known way to prove that SHA-256 is collision-resistant.

The RSA spec, derived from RFC 2313 [33], defines encryption and signature operations as modular exponentiation on keys made of Dafny's ideal integers. The key-generation spec requires that the key be made from two random primes.

To implement these crypto primitives, we built a BigNum library. It implements arbitrary-precision integers using arrays of 32-bit words, providing operations like division and modulo needed for RSA. BigRat extends it to rationals, needed for differential privacy.

Lemma 12 BIGNUM/BIGRAT CORRECTNESS. *Each BigNum/BigRat operation produces a value representing the correct infinite-precision integer or real number.*

6.3 DafnyCC-generated code

Since the DafnyCC compiler sits outside our TCB, we have to verify the assembly language code it generates. This verification rests on several invariants maintained by all DafnyCC-generated code:

Lemma 13 TYPE SAFETY. *The contents of every value and heap object faithfully represent the expected contents according to Dafny's type system, so that operations on these values never cause run-time type errors.*

Lemma 14 ARRAY BOUNDS SAFETY. *All array operations use an index within the bounds of the array.*

Lemma 15 TRANSITIVE STACK SAFETY. *When calling a method, enough stack space remains for all stack operations in that method and those it in turn calls.*

Dafny is a type-safe language, but we cannot simply assume that DafnyCC preserves Dafny's type safety. Thus, we must prove type safety at the assembly language level by establishing typing invariants on all data structures that represent Dafny values. For example, all pointers in data structures point only to values of the expected type, and arbitrary integers cannot be used as pointers. These typing invariants are maintained throughout the Ironclad assembly language code (they appear in nearly all loop invariants, preconditions, and postconditions). In contrast to the original Verve OS [65], Ironclad does not rely on an external typed assembly language checker to check compiled code; this gives Ironclad the advantage of using a single verification process for both hand-written assembly language code and compiled code, ensuring that there are no mismatches in the verification process.

Lemma 16 HIGH-LEVEL PROPERTY PRESERVATION. *Every method proves that output stack state and registers satisfy the high-level Dafny postconditions given the high-level Dafny preconditions.*

DafnyCC maintains all Dafny-level annotations, including preconditions, postconditions, and loop invariants. Furthermore, it connects these high-level annotations to low-level stack and register values, so that the operations on stack and register values ultimately satisfy the Dafny program's high-level correctness theorems.

6.4 Maintaining OS internal invariants

Although Ironclad builds on the original Verve OS [65], we made many modifications to the Verve code to accommodate DafnyCC, the late launch process and the IOMMU (§6.1), the TPM (§2.3), segmentation, and other aspects of Ironclad. Thus, we had to prove that these modifications did not introduce any bugs into the Verve code.

Lemma 17 OPERATING SYSTEM INVARIANTS. *All operating system data structure invariants are maintained.*

Lemma 18 GARBAGE COLLECTION CORRECTNESS. *The memory manager's representation of Dafny objects correctly represents the high-level Dafny semantics.*

We modified the original Verve copying garbage collector's object representation to accommodate DafnyCC-generated code. This involved reproving the GC correctness lemma: that the GC always maintains correct object data, and never leaves dangling pointers, even as it moves objects around in memory. Our modification initially contained a design flaw in the object header word: we accidentally used the same bit pattern to represent two different object states, which would have caused severe and difficult-to-debug memory corruption. Verification found the error in seconds, before we ran the new GC code.

7 Experiences and Lessons Learned

In this section, we describe our experiences using modern verification tools in a large-scale systems project, and the solutions we devised to the problems we encountered.

7.1 Verification automation varies by theory

Automated theorem provers like Z3 support a variety of theories: arithmetic, functions, arrays, etc. We found that Z3 was generally fast, reliable, and completely automated at reasoning about addition, subtraction, multiplication/division/mod by small constants, comparison, function declarations, non-recursive function definitions, sequence/array subscripting, and sequence/array updates. Z3 sometimes needed hints to verify sequence concatenation, forall/exists, and recursive function definitions, and to maintain array state across method invocations.

Unfortunately, we found Z3's theory of nonlinear arithmetic to be slow and unstable; small code changes often caused unpredictable verification failures (§7.2).

7.2 Verification needs some manual control

As discussed in §1, verification projects often avoid automated tools for fear that such tools will be unstable and/or too slow to scale to large, complex systems. Indeed, we encountered verification instability for large formulas and nonlinear arithmetic. Nevertheless, we were able to address these issues by using modular verification (§3.5), which reduced the size of components to be verified, and two additional solutions:

Opaque functions. Z3 may unwrap function definitions too aggressively, each time obtaining a new fact, often leading to timeouts for large code. To alleviate this, we modified Dafny so a programmer can designate a function as *opaque*. This tells the verifier to ignore the body, except in places where the programmer explicitly indicates.

Nonlinear math library. Statements about nonlinear integer arithmetic, such as $\forall x, y, z : x(y+z) = xy + xz$, are not, in general, decidable [17]. So, Z3 includes heuristics for reasoning about them. Unfortunately, if a complicated

method includes a nonlinear expression, Z3 has many options for applicable heuristics, leading to instability.

Thus, we disable Z3's nonlinear heuristics, except on a few files where we prove simple, fundamental lemmas, such as $(x > 0 \wedge y > 0) \Rightarrow xy > 0$. We used those fundamental lemmas to prove a library of math lemmas, including commutativity, associativity, distributivity, GCDs, rounding, exponentiation, and powers of two.

7.3 Existing tools make simple specs difficult

To enhance the security of Ironclad Apps, we aim to minimize our TCB, particularly the specifications.

Unfortunately, Dafny's verifier insists on proving that, whenever one function invokes another, the caller meets the callee's pre-conditions. So, the natural spec for SHA,

```
function SHA(bits:seq<int>):seq<int>
  requires |bits|<power2(64);
{ .... }
function SHA_B(bytes:seq<int>):seq<int>
{ SHA(Bytes2Bits(bytes)) }
```

has a problem: the call from `SHA_B` to `SHA` may pass a bit sequence whose length is $\geq 2^{64}$.

We could fix this by adding

```
requires |bytes|<power2(61);
```

to `SHA_B`, but this is insufficient because the verifier needs help to deduce that 2^{61} bytes is 2^{64} bits. So we would also have to embed a mathematical proof of this in the body of `SHA_B`, leading to a bloated spec.

Automatic requirements. Our solution is to add *automatic requirement propagation* to Dafny: A spec writer can designate a function as `autoReq`, telling Dafny to automatically add pre-conditions allowing it to satisfy the requirements of its callees. For instance, if we do this to `SHA_B`, Dafny gives it the additional pre-condition:

```
requires |Bytes2Bits(bytes)|<power2(64);
```

This makes the spec verifiable despite its brevity.

Premium functions. Our emphasis on spec simplicity can make the implementor's job difficult. First, using `autoReq` means that the implementor must satisfy a pile of ugly, implicit, machine-generated pre-conditions everywhere a spec function is mentioned. Second, the spec typically contains few useful post-conditions because they would bloat the spec. For instance, `SHA` does not state that its output is a sequence of eight 32-bit words.

We thus introduce a new discipline of using *premium* functions in the implementation. A premium function is a variant of a spec function optimized for implementation rather than readability. More concretely, it has simpler-to-satisfy pre-conditions and/or more useful post-conditions. For instance, instead of the automatically-generated pre-conditions, we use the tidy pre-conditions we wanted to write in the spec but didn't because we didn't want to prove them sufficient. For instance, we could use

```
  requires |bits|<power2(61);
  ensures IsWordSeqOfLen(hash, 8);
```

as the signature for the premium version of `SHA_B`.

7.4 Systems often use bounded integer types

Dafny only supports integer types `int` and `nat`, both representing unbounded-size values. However, nearly all of our code concerns bounded-size integers such as bits, bytes, and 32-bit words. This led to many more annotations and proofs than we would have liked. We have provided this feedback to Dafny's author, who consequently plans to add refinement types.

7.5 Libraries should start with generality

Conventional software development wisdom is to start with simple, specific code and generalize only as needed, to avoid writing code paths which are not exercised or tested. We found this advice invalid in the context of verification: instead, it is often easier to write, prove, and use a more-general statement than the specific subset we actually need. For example, rather than reason about the behavior of shifting a 32-bit integer by k bits, it is better to reason about shifting n-bit integers k bits. Actual code may be limited to $n = 32$, but the predicates and lemmas are easier to prove in general terms.

7.6 Spec reviews are productive

Independent spec reviews (§3.3) caught multiple human mistakes; for instance, we caught three bugs in the segmentation spec that would have prevented our code from working the first time. Similarly, we found two bugs in the SHA-1 spec; these were easily detected, since the spec was written to closely match the text of the FIPS spec [50].

To our knowledge, only three mistakes survived the review process, and all three were liveness, not security, bugs in the TPM spec: Code written against the original spec would, under certain conditions, wait forever for an extra reply byte which the TPM would never send.

Also, our experience was consistent with prior observations that the act of formal specification, even before verification, clarifies thinking [38]. This discipline shone especially in specifying hardware interfaces, such as x86 segmentation behavior. Rather than probing the hardware's behavior with a code-test-debug cycle, specification required that we carefully extract and codify the relevant bits of Intel's Byzantine documentation. This led to a gratifying development experience in which our code worked correctly the first time we ran it.

7.7 High-level tools have bugs

One of our central tenets is that verification should be performed on the low-level code that will actually run, not the high-level code it is compiled from. This is meant to reduce bugs by removing the compiler from the TCB. We

found that this is not just a theoretical concern; we discovered *actual* bugs that this approach eliminates.

For example, when testing our code, we found a bug in the Dafny-to-C# compiler that suppressed calls to methods with only ghost return values, even if those methods had side effects. Also, we encountered a complex bug in the translation of while loops that caused the high-level Dafny verifier to report incorrect code as correct. Finally, verifying at the assembly level caught multiple bugs in DafnyCC, from errors in its variable analysis and register allocator to its handling of calculational proofs.

8 Evaluation

We claim that it is feasible to engineer apps fully verified to adhere to a security-sensitive specification. We evaluate this claim by measuring the artifacts we built and the engineering effort of building them.

8.1 System size

Table 1 breaks down the components of the system. It shows the size of the various specs, high-level implementation code, and proof statements needed to convince the verifier that the code meets the specs. It also shows the amount of verifiable assembly code, most generated by DafnyCC but some written by hand. Overall, Ironclad consists of 3,546 lines of spec, plus 7K lines of implementation that compile to 42K assembly instructions. Verifying the system takes 3 hours for functional-correctness properties and an additional 21 hours for relational properties.

Most importantly, our specs are small, making manual spec review feasible. Altogether, all four apps have 3,546 SLOC of spec. The biggest spec components are hardware and crypto. Both these components are of general use, so we expect spec size to grow slowly as we add additional apps.

Our ratio of implementation to spec is 2:1, lower than we expected. One cause for this low ratio is that much of the spec is for hardware, where the measured implementation code is just drivers and the main implementation work was done by the hardware manufacturers. Another cause is that we have done little performance optimization, which typically increases this ratio.

Our ratio of proof annotation to implementation, 4.8:1, compares favorably to seL4's ∼20:1. We attribute this to our use of automated verification to reduce the burden on developers. Note also that the ratio varies across components. For instance, the core system and math libraries required many proofs to establish basic facts (§7.2); thanks to this work, higher-level components obtained lower ratios. Since these libraries are reusable, we expect the ratio to go down further as more apps reuse them.

Figure 6 shows line counts for our tools. The ones in our TCB have 15,302 SLOC. This is much less than the

Component	Spec (SLOC)	Impl (SLOC)	Proof (SLOC)	Asm (LOC)	Boogie time (s)	SymDiff time (s)
Specific apps:						
PassHash	32	81	193	447	158	6434
TrInc	78	232	653	1292	438	9938
Notary	38	140	307	663	365	14717
DiffPriv	444	586	1613	3523	891	21822
Ironclad core:						
App common	43	64	119	289	210	0
SHA-1,-256	420	574	3089	6049	698	0
RSA	492	726	4139	3377	1405	9861
BigNum	0	1606	8746	7664	2164	0
UDP/IP stack	0	135	158	968	227	4618
Seqs and ints	177	312	4669	1873	791	888
Datatypes	0	0	0	5865	1827	0
Core math	72	206	3026	476	571	0
Network card	0	336	429	2126	199	3547
TPM	296	310	531	2281	417	0
Other HW	90	324	671	2569	153	3248
Modified Verve:						
CPU/memory	900	643	2131	260	67	0
I/O	464	410	1126	1432	53	1533
GC	0	286	1603	412	92	0
Total	3546	6971	33203	41566	10726	76606

Table 1: **System Line Counts and Verification Times.** *Asm LOC includes both compiled Dafny and hand-written assembly.*

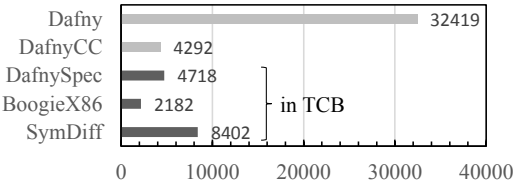


Figure 6: **Tool Line Counts.**

32,419 SLOC in the original Dafny-to-C# compiler, let alone the code for the C# compiler. Our DafnyCC tool is 4,292 SLOC and depends on Dafny as well, but as discussed earlier (§3.4), it is not in the TCB.

8.2 Developer effort

Previous work based on interactive proof assistants showed that the costs can be quite high [48]. In contrast, our experience suggests that automated provers reduce the burden to a potentially tolerable level. Despite learning and creating new tools, as well as several major code refactorings, we constructed the entire Ironclad system with under three person-years of effort.

8.3 Performance

Finally, although performance was not one of our goals, we evaluate the performance of our apps to demonstrate how much more work lies ahead in optimization. For these experiments, we use as our server an HP Compaq 6005 Pro PC with a 3-GHz AMD Phenom II X3 CPU, 4 GB of RAM, and a Broadcom NetXtreme Gigabit Eth-

Operation	Dominant step	Ironclad		Unverified		Slowdown
Notary `notarize`	Compute RSA signature	934 ms	±0 ms	8.88 ms	±0.68 ms	105
TrInc `create`	Compute reciprocal of RSA modulus	4.96 ms	±0.03 ms	866 µs	±507 µs	5.72
TrInc `advance`	Compute RSA signature	1.01 s	±0.00 s	12.1 ms	±0.2 ms	83.6
PassHash `hash`	Compute SHA-256 hash	276 µs	±10 µs	159 µs	±3 µs	1.73
DiffPriv `initialize_db`	None, just network overhead	168 µs	±2 µs	155 µs	±3 µs	1.08
DiffPriv `add_row`	Decrypt RSA-encrypted row	944 ms	±17 ms	8.85 ms	±0.06 ms	107
DiffPriv `query`	Compute noise with BigInts	126 ms	±19 ms	668 µs	±36 µs	189

Table 3: **App benchmarks.** *Latency of different request types, as seen by a client on the same network switch, for Ironclad Apps and unverified variants written in C#. Ranges shown are 95% confidence intervals for the mean.*

Op	Param	Ironclad	OpenSSL	C#/.NET
TPM GetRandom	256b	39 µs/B	*–	*–
RSA KeyGen	1024b	39.0 s	*12 ms	*181 ms
RSA Public	1024b	35.7 ms	204 µs	65 µs
RSA Private	1024b	858 ms	4.03 ms	1.40 ms
SHA-256	256B	26 ns/B	13 ns/B	24 ns/B
SHA-256	8192B	13 ns/B	10 ns/B	7.6 ns/B

Table 2: **Crypto microbenchmarks.** **Only Ironclad uses the TPM for KeyGen.*

ernet NIC. As our client, we use a Dell Precision T7610, with a 6-core 2.6-GHz Xeon E5-2630 CPU, 32 GB of RAM, and an Intel 82579LM Gigabit Ethernet NIC. The two are connected to the same gigabit switch.

Table 2 shows the latency and bandwidth of various low-level crypto and TPM operations; these operations constitute the dominant cost of app initialization and/or app operations. Table 2 also shows the corresponding times for C#/.NET code on Windows 7 and OpenSSL on Ubuntu. The results show that our RSA library is about two orders of magnitude slower than unverified variants, and our SHA-256 code approaches within 30%.

This poor performance is not fundamental to our approach. Indeed, verification provides a safety net for aggressive optimizations. For example, we extended DafnyCC to support directed inlining, applied this to the code for SHA-256, then performed limited manual optimization on the resulting verifiable assembly. This more than doubled our code's performance, bringing us within 30% of OpenSSL. Along the way, we had no fear of violating correctness; indeed, the verifier caught several bugs, e.g., clobbering a live register. We used the same technique to manually create a verified assembly-level unrolled add function for BigIntegers. Similarly, verification helped to correctly move our multi-precision integer library from immutable sequences to mutable arrays, making it 1000× faster than the first version. Many optimization opportunities remain, such as unrolled loops, inlined procedures, and arithmetic using the Chinese remainder theorem and Montgomery form.

Next, we show the performance of high-level operations. To compare Ironclad's performance to unverified servers, we wrote unverified variants of our apps in C# using the .NET Framework. We run those apps on the server on Windows 7.

Table 3 shows the results. We measure various operations from the client's perspective, counting the time between sending a request and receiving the app's reply. For each operation, we discard the first five results and report the mean of the remaining 100 results; we also report the 95% confidence interval for this mean. We use 1,024-bit RSA keys, 32-byte hashes for `notarize` and `advance`, 12-byte passwords and 16-byte salts for `hash`, 20-byte nonces and four-column rows for `add_row`, and a 19-instruction mapper for `query`.

The results are generally consistent with the microbenchmark results. Slowdowns are significant for operations whose dominant component involves an RSA key operation (`notarize`, `advance`, `add_row`), and lower but still substantial for those involving SHA-256 (`hash`) and big-integer operations (`create` and `query`). The `initialize_db` operation, which involves no cryptographic operations and essentially just involves network communication, incurs little slowdown.

9 Limitations and Future Work

As with any verified system, our guarantees only hold if our specs, both of hardware and apps, are correct. While we strive to keep the specs minimal and take additional steps to add assurance (§3.3), this is the most likely route for errors to enter the system.

We also rely on the correctness of our verification tools, namely our Dafny spec translator, SymDiff, Boogie, and Z3. Unfortunately, these tools do not currently provide proof objects that can be checked by a small verifier, so they all reside in our TCB. Fortunately, our spec translator is tiny, and Boogie and Z3 are extensively tested and used by dozens of projects, including in production systems. In practice, we did not encounter any soundness bugs in these tools, unlike the untrusted, higher-level tools we employed (§7.7).

At present, we do not model the hardware in enough detail to prove the absence of covert or side channels that may exist due to timing or cache effects, but prior work [48] suggests that such verification is feasible.

Currently, we prove the functional correctness and non-interference of our system, but our proofs could be extended in two directions that constitute ongoing work: proving liveness, and connecting our guarantees to even higher-level cryptographic protocol correctness proofs. For example, we want to explicitly reason about probability distributions to show that our use of cryptographic primitives creates a secure channel [6, 9].

With Ironclad, we chose to directly verify all of our code rather than employing verified sandboxing. However, our implementation supports provably correct page table usage and can safely run .NET code, so future work could use unverified code as a subroutine, checking its outputs for desired properties. Indeed, type safety allows code to safely run in kernel mode, to reduce kernel-user mode crossings.

10 Related Work

Trusted Computing. As discussed in §1, Trusted Computing has produced considerable research showing how to identify code executing on a remote machine [53]. However, with a few exceptions, it provides little guidance as to how to assess the security of that code. Property-based attestation [58] shifts the problem of deciding if the code is trustworthy from the client to a trusted third party, while semantic attestation attests to a large software stack—a traditional OS and managed runtime—to show that an app is type safe [29]. The Nexus OS [60] attests to an unverified kernel, which then provides higher-level attestations about the apps it runs. In general, verification efforts in the Trusted Computing space have focused primarily on the TPM's protocols [11, 16, 27, 44] rather than on the code the TPM attests to.

Early security kernels. Issued in 1983, the DoD's "Orange Book" [18] explicitly acknowledged the limitations of contemporary verification tools. The highest rating (A1) required a formal specification of the system but only an informal argument relating the code to the specification. Early efforts to attain an A1 rating met with mixed success; the KVM/370 project [25] aimed for A1, but, due in part to inadequacies of the languages and tools available, settled for C2. The VAX VMM [34] did attain an A1 rating but could not verify that their implementation satisfied the spec. Similar caution applies to other A1 OSes [22, 59].

Recent verified kernels. The seL4 project [35, 36, 48] successfully verified a realistic microkernel for strong correctness properties. Doing so required roughly 200,000 lines of manual proof script to verify 8,700 lines of C code using interactive theorem proving (and 22 person-years); Ironclad's use of automated theorem proving reduces this manual annotation overhead, which helped to reduce the effort required (3 person-years). seL4 has focused mainly on kernel verification; Ironclad contains the small Verve verified OS, but focuses more on library (e.g. BigNum/RSA), driver (e.g. TPM), and application verification in order to provide whole-system verification. seL4's kernel is verified, but can still run unverified code outside kernel mode. Ironclad currently consists entirely of verified code, but it can also run unverified code (§9). Both seL4 and Ironclad verify information flow.

Recent work by Dam et al. [14] verifies information-flow security in a simple ARM separation kernel, but the focus is on providing a strict separation of kernel usages among different security domains. This leaves other useful security properties, including functional correctness of the OS and applications, unverified.

While seL4 and Ironclad Apps run on commodity hardware, the Verisoft project [2] aimed for greater integration between hardware and software verification, building on a custom processor. Like seL4, Verisoft required >20 person-years of effort to develop verified software.

Differential privacy. Many systems implement differential privacy, but none provide end-to-end guarantees about their implementations' correctness. For instance, Barthe et al. describe *Certipriv* [8], a framework for mechanically proving algorithms differentially private, but do not provide an executable implementation of these algorithms. As a consequence, implementations have vulnerabilities; e.g., Mironov [46] demonstrated an attack that affected PINQ [43], Airavat [56], Fuzz [55], and GUPT [47].

11 Conclusion

By using automated tools, we have verified full-system, low-level, end-to-end security guarantees about Ironclad Apps. These security guarantees include non-trivial properties like differential privacy, which is notoriously difficult to get right. By writing a compiler from Dafny to verified assembly language, we verified a large suite of libraries and applications while keeping our tool and specification TCB small. The resulting system, with ∼6500 lines of runnable implementation code, took ∼3 person-years to verify. Beyond small, security-critical apps like Ironclad, verification remains challenging: assuming ∼2000 verified LOC per person-year, a fully verified million-LOC project would still require ∼100s of person-years. Fortunately, the tools will only get better, so we expect to see full-system verification scale to larger systems and higher-level properties in the years to come.

Acknowledgments

We thank Jeremy Elson, Cedric Fournet, Shuvendu Lahiri, Rustan Leino, Nikhil Swamy, Valentin Wuestholz, Santiago Zanella Beguelin, and the anonymous reviewers for their generous help, guidance, and feedback. We are especially grateful to our shepherd Gernot Heiser, whose insightful feedback improved the paper.

References

[1] Advanced Micro Devices. AMD64 Architecture Programmer's Manual. AMD Publication no. 24593 rev. 3.22, 2012.

[2] E. Alkassar, M. A. Hillebrand, D. C. Leinenbach, N. W. Schirmer, A. Starostin, and A. Tsyban. Balancing the load: Leveraging semantics stack for systems verification. *Automated Reasoning*, 42(2–4):389–454, 2009.

[3] B. Balacheff, L. Chen, S. Pearson, D. Plaquin, and G. Proudler. *Trusted Computing Platforms – TCPA Technology in Context*. Prentice Hall, 2003.

[4] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. *Proceedings of Formal Methods for Components and Objects (FMCO)*, 2006.

[5] G. Barthe, C. Fournet, B. Gregoire, P.-Y. Strub, N. Swamy, and S. Z. Beguelin. Probabilistic relational verification for cryptographic implementations. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*, Jan. 2014.

[6] G. Barthe, B. Grégoire, S. Heraud, and S. Zanella-Béguelin. Computer-aided security proofs for the working cryptographer. In *Proceedings of IACR CRYPTO*, 2011.

[7] G. Barthe, B. Grégoire, Y. Lakhnech, and S. Zanella-Béguelin. Beyond provable security. Verifiable IND-CCA security of OAEP. In *Proceedings of CT-RSA*, 2011.

[8] G. Barthe, B. Köpf, F. Olmedo, and S. Zanella-Béguelin. Probabilistic relational reasoning for differential privacy. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*, 2012.

[9] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P. Strub. Implementing TLS with verified cryptographic security. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2013.

[10] B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, and G. Heiser. Timing analysis of a protected operating system kernel. In *Proceedings of the IEEE Real-Time Systems Symposium*, 2011.

[11] D. Bruschi, L. Cavallaro, A. Lanzi, and M. Monga. Replay attack in TCG specification and solution. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2005.

[12] C. Cachin. Distributing trust on the Internet. In *Proceedings of the IEEE/IFIP Conference on Dependable Systems and Networks (DSN)*, 2001.

[13] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *Proceedings of the Conference on Theorem Proving in Higher Order Logics*, 2009.

[14] M. Dam, R. Guanciale, N. Khakpour, H. Nemati, and O. Schwarz. Formal verification of information flow security for a simple ARM-based separation kernel. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2013.

[15] N. A. Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*, 2008.

[16] A. Datta, J. Franklin, D. Garg, and D. Kaynar. A logic of secure systems and its application to trusted computing. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2009.

[17] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[18] Department of Defense. *Trusted Computer System Evaluation Criteria*. National Computer Security Center, 1983.

[19] C. Dwork. Differential privacy. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*, 2006.

[20] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Proceedings of the IACR Theory of Cryptography Conference (TCC)*. 2006.

[21] R. W. Floyd. Assigning meanings to programs. In *Symposium on Applied Mathematics*, 1967.

[22] L. J. Fraim. SCOMP: A solution to the multilevel security problem. *Computer*, 16:26–34, July 1983.

[23] A. Ghosh, T. Roughgarden, and M. Sundararajan. Universally utility-maximizing privacy mechanisms. In *Proceedings of the ACM Symposium on the Theory of Computing (STOC)*, 2009.

[24] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1982.

[25] B. D. Gold, R. R. Linde, and P. F. Cudney. KVM/370 in retrospect. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1984.

[26] S. Gulwani, K. K. Mehra, and T. Chilimbi. SPEED: Precise and efficient static estimation of program computational complexity. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*, 2009.

[27] S. Gürgens, C. Rudolph, D. Scheuermann, M. Atts, and R. Plaga. Security evaluation of scenarios based on the TCG's TPM specification. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, 2007.

[28] S. Haber and W. S. Stornetta. How to time-stamp a digital document. *Journal of Cryptology*, 3, 1991.

[29] V. Haldar, D. Chandra, and M. Franz. Semantic remote attestation: a virtual machine directed approach to trusted computing. In *Proceedings of the Conference on Virtual Machine Research*, 2004.

[30] G. Heiser, L. Ryzhyk, M. von Tessin, and A. Budzynowski. What if you could actually trust your kernel? In *Hot Topics in Operating Systems (HotOS)*, 2011.

[31] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[32] Intel Corporation. Intel Trusted Execution Technology – Measured Launched Environment Developer's Guide. Document number 315168-005, June 2008.

[33] B. Kaliski. PKCS #1: RSA cryptography specifications version 1.5. RFC 2313, Mar. 1998.

[34] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn. A retrospective on the VAX VMM security kernel. *IEEE Transactions on Software Engineering*, 17(11):1147–1165, Nov. 1991.

[35] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1), 2014.

[36] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, M. Norrish, R. Kolanski, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[37] S. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebelo. SymDiff: A language-agnostic semantic diff tool for imperative programs. In *Proceedings of Computer Aided Verification (CAV)*, July 2012.

[38] L. Lamport. *Specifying Systems: The TLA+ Languange and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[39] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, 2010.

[40] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. TrInc: Small trusted hardware for large distributed systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[41] M. Mazurek, S. Komanduri, T. Vidas, L. Bauer, N. Christin, L. F. Cranor, P. Kelley, R. Shay, and B. Ur. Measuring password guessability for an entire university. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2013.

[42] F. Mckeen, I. Alexandrovich, A. Berenzon, C. Rozas, H. Shafi, V. Shanbhogue, and U. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[43] F. McSherry. Privacy integrated queries: An extensible platform for privacy-preserving data analysis. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2009.

[44] J. Millen, J. Guttman, J. Ramsdell, J. Sheehy, and B. Sniffen. Analysis of a measured launch. Technical Report 07-0843, The MITRE Corporation, June 2007.

[45] G. L. Miller. Riemann's hypothesis and tests for primality. *Journal of Computer and System Sciences*, 13(3), 1976.

[46] I. Mironov. On significance of the least significant bits for differential privacy. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2012.

[47] P. Mohan, A. Thakurta, E. Shi, D. Song, and D. E. Culler. GUPT: Privacy preserving data analysis

made easy. In *ACM International Conference on Management of Data (SIGMOD)*, 2012.

[48] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. seL4: From general purpose to a proof of information flow enforcement. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2013.

[49] National Institute of Standards and Technology. The keyed-hash message authentication code (HMAC), 2012. FIPS PUB 198-1.

[50] National Institute of Standards and Technology. Secure hash standard (SHS), 2012. FIPS PUB 180-4.

[51] National Vulnerability Database. Heartbleed bug. CVE-2014-0160 http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0160, Apr. 2014.

[52] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2002.

[53] B. Parno, J. M. McCune, and A. Perrig. *Bootstrapping Trust in Modern Computers*. Springer, 2011.

[54] M. O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1), 1980.

[55] J. Reed and B. C. Pierce. Distance makes the types grow stronger: A calculus for differential privacy. In *Proceedings of the ACM International Conference on Functional Programming*, 2010.

[56] I. Roy, S. T. V. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: Security and privacy for MapReduce. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2010.

[57] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.

[58] A.-R. Sadeghi and C. Stueble. Property-based attestation for computing platforms: Caring about properties, not mechanisms. In *Proceedings of the Workshop on New Security Paradigms (NSPW)*, 2004.

[59] W. Shockley, T. Tao, and M. Thompson. An overview of the GEMSOS class A1 technology and application experience. In *Proceedings of the National Computer Security Conference*, Oct. 1988.

[60] E. G. Sirer, W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, and F. B. Schneider. Logical attestation: An authorization architecture for trustworthy computing. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[61] G. Smith. Principles of secure information flow analysis. *Malware Detection*, pages 297–307, 2007.

[62] S. G. Stubblebine and P. F. Syverson. Fair on-line auctions without special trusted parties. In *Proceedings of Financial Cryptography*, 1999.

[63] Trusted Computing Group. Trusted Platform Module Main Specification. Version 1.2, Revision 116, 2011.

[64] Wave Systems Corp. Trusted Computing: An already deployed, cost effective, ISO standard, highly secure solution for improving Cybersecurity. http://www.nist.gov/itl/upload/Wave-Systems_Cybersecurity-NOI-Comments_9-13-10.pdf, 2010.

[65] J. Yang and C. Hawblitzel. Safe to the last instruction: Automated verification of a type-safe operating system. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2010.

[66] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.