Avraham “Abe” Bernstein | Master S/W Engineer | Abbrev CV

Represented by Tk Open Systems Ltd.

Version: 0.4.1-abbrev-tkos

Last update: 2017-09-24T01:47:21Z

0.1 Contact Info & Links

email/skype: Avraham.Bernstein+tkos@gmail.com

geolocation: Jerusalem Israel, **tz:** UTC +02:00/+03:00 [winter/summer]

tel-IL-mobile/whatsapp: +972.54.641-0955

tel-US-mobile: +1.845.402-0023

cv-full: HTML, DOCX, PDF

cv-abbrev: HTML, DOCX, PDF **[this file]**

1.0 Summary

I am an experienced computer scientist and S/W architect. I have devised innovative solutions to many S/W problems for a wide range of fields, including

- cybersecurity
- cryptography
- bioinformatics
- factory automation
- VLSI CPU design
- telecommunications
- blind vision
- accessibility
- transportation vehicle route guidance
- automated testing

I have worked for a number of organizations, large and small, and helped them realize improvements in their product performance, often putting them in the front rank in their field. I have acquired expert knowledge in a number of fields, often liaising with noted experts, and have been able to quickly apply this knowledge to improve the competitive position of the companies and their products. I have a keen interest in computer languages, both practical and theoretical. I have created a number of domain specific languages (DSL) that were instrumental in greatly simplifying seemingly intractable problems.

In order to understand how I design S/W see the following appendices:

- Programming Language Preferences and Musings
- Domain Specific Languages
- How To Write Correct, Maintainable, Secure, and Easy-to-Test Code

2.0 Work Experience

2017-present: Cybersecurity Consultant

@Self-Employed, Jerusalem:

Keys: cybersecurity, architect, algorithms, obfuscation, compiler, **C/C++** , javascript, WASM

1. I am developing an obfuscating compiler for C/C++ and for Web Assembly (WASM). Still in stealth mode.
2. I am a security mentor for the Jerusalem Mass Challenge start-up hub.

2011-17: S/W Architect & Developer: Cybersecurity: OTT Internet Pay TV System

@Viaccess-Orca, Ra'anana - a subsidiary of Orange FR, and @Discretix/SansaSecurity, Netanya - now merged into ARM:

Keys: cybersecurity, DRM, architect, algorithms, anti-reverse engineering, obfuscation, LLVM compiler, cryptography, **C/C++** , TCL, Python, bash, Android root detection, Linux, ELF edit, IOS

1. I architected and implemented anti-reverse engineering and obfuscation programming frameworks and libraries in C/C++ for their DRM protected movie player application that ran on Android and IOS devices.
2. The challenges of implementing obfuscation are that (1) the other programmers should not be concerned about it because their focus must be on writing correct code, and (2) the resulting increase in size and reduction in run-time speed must not noticeably reduce the usability/functionality of the application. In general the aim of obfuscation is to provide "good enough security" that will deter 95% of potential attackers, and when combined with regular application updates will force an attacker to begin his next reverse engineering attempt from scratch.
3. I developed a post processor to obfuscate the resulting binary object ELF files.
4. I developed a light weight obfuscated cryptographic library implemented as a H file using inline functions so that every module that included it had its own private copy of the library with a module specific randomized implementation which prevented an attack against a single core cryptographic module that could potentially subvert the whole application.
5. I developed an Android root detection mechanism using fuzzy logic techniques.
6. I developed a background watchdog security thread to dynamically ensure that the binary code had not been tampered with.
7. All secure code modules were implemented as native libraries written in C/C++.
8. Offline utilities and build scripts were written in bash, Python, and TCL.
9. My typical development methodology was to first build a prototype for desktop Linux, secondly as a standalone CLI application on the target device, and finally to incorporate the source code into the full application on the target device. Whenever possible I preferred to test on virtual machines.

10. I was responsible for the purchase decisions and usage policy of 3rd party obfuscation and cryptographic utilities and libraries.
11. **At the end of my 6 year tenure there were 40M subscribers, and no security breaches.**

2016-16: Cybersecurity Consultant: Protection of a Small Business with Extremely High Security Concerns

@Anonymous, Jerusalem: See details.

Keys: cybersecurity, privacy, anonymity, WordPress, static web site, Cloudflare, Windows, Android, Google Docs, Google Drive

2010-11: S/W Architect & Developer: Transportation: Urban Traffic Vehicle Route Guidance Algorithms

@TeleQuest, Jerusalem: See details.

Keys: urban vehicle route guidance, architect, algorithms, Java, AWS

2009-09: S/W Architect & Developer: Bioinformatics: PCR Algorithm

@Syntezza, Jerusalem: See details.

Keys: bioinformatics, PCR, architect, algorithms, mathematical programming, **C** , Python

2004-09: Cybersecurity Researcher for a CA Satellite Pay TV System

@Cisco-NDS, Jerusalem: See details.

Keys: cybersecurity, DRM, algorithms, cryptography, anti-reverse engineering, obfuscation, LLVM compiler, VM, QEMU, RPC, automated testing, S/W quality, **C/C++** , TCL, Python, Linux, bash, Win32

2002-03: S/W Architect & Developer: Accessibility: Enabled Blind to “See” Maps

@Virtouch, Jerusalem: See details.

Keys: accessibility, blind, architect, GIS, MapML, HTML, SVG, javascript, XSLT, XML Schema, XSLT, **C** , TCL

1999-2002: S/W Architect & Developer: Network: Utilities for a “Wireless” Cable Modem and Router System

@Vyyo, Jerusalem: See details.

Keys: network, architect, algorithms, SNMP, SNMP-agent, NMS, automated testing, **C** , TCL, embedded

2001-01: S/W Architect & Developer: Network: Network Management System (NMS) for a FSO Device

@MRV-Jolt, Jerusalem: See details.

Keys: network, architect, SNMP, SNMP-agent, NMS, Java, **C** , TCL

2001-01: Consultant: Network Management System (NMS) for a Cable Modem & Gateway System

@Foxcom - One Path Networks, Jerusalem: See details.

Keys: SNMP, NMS

2000-01: S/W Developer: Communications: Win32 Asynchronous TCP/IP DLL for a Visual Basic Project

@Inex-Zamir, Jerusalem: See details.

Keys: TCP/IP communications, **C** , Visual Basic, Win32, soft real-time

1998-99: S/W Architect & Developer: Compiler: GCC Compiler Port for a 128-Core Stack Machine

@Fourfold, Jerusalem: See details.

Keys: C compiler, gcc, architect, FORTH, **C/C++** , TCL, LISP

1997-98: S/W Architect & Developer: Factory Automation: Conoscopic Interferometer Workstation

@Newport-Optimet: See details.

Keys: measurement workstation, architect, DSL, **C** , TCL, OpenGL, Win32, soft real-time

1996-97: Lecturer: Win32 Internals Course

@M.E.R., Jerusalem:

Keys: lecturer, Win32, **C**

1996-96: Consulting S/W Engineer: Win32 Improve Performance of a Soft Real-Time Biofeedback Application

@UltraMind, Jerusalem: See details.

Keys: Win32, soft real-time, **C**

1996-96: Consulting S/W Engineer: Win32 Device Driver for a Frame Grabber

@Cefar, Jerusalem:

Keys: Win32, soft real-time

1995-96: S/W Architect & Developer: US DOD Mil-Spec Automated Testing: Night Hawk Fire Control System

@Elbit-Elop, Rechovot: See details.

Keys: automated testing, mil-spec, architect, DSL, **C/C++** , BASIC compiler, lex/yacc, Win32, soft real-time

1995-95: Lecturer: Introductory University Computer Science Course on Database Theory

@Michlala College Bayit Vegan, Jerusalem:

Keys: lecturer, database, SQL

1991-94: S/W Architect & Developer: VLSI: Simulator & S/W Toolchain For DSPG PINE CPU

@DSP Group, Givat Shmuel: See details.

Keys: VLSI simulator, S/W Development Toochain, architect, DSL, **C/C++** , lex/yacc, assembly, Win32

1989-91: S/W Architect & Developer: Factory Automation: Shop Floor Production Control (SFPC) System: BARI II

@Digital Equipment Corporation (DEC), Herzliya, for @Iscar, Tefen: See details.

Keys: factory automation SFPC, architect, DSL, Pascal, SQL, VAX/VMS

1988-88: S/W Architect & Developer: Accessibility: Quadriplegic PC Accessibility

@Cubital, Herzliya - a charity project funded by the company and their CEO Itzhak Pomerantz in cooperation with the Beit Levinson Rehabilitation Hospital, and the IDF Rehabilitation Unit: See details.

Keys: accessibility, Prolog, PC-DOS

1987-88: S/W Developer & VAX/VMS Sysadmin: 3D Printer: Solider

@Cubital, Herzliya: See details.

Keys: 3D printing, **C** , sysadmin, VAX/VMS

1986-87: S/W Developer: Soft Real-Time RS232 Z80 Communication Driver: Data Collection & Access Control Terminal

@Elde, Jerusalem:

Keys: data collection terminal, **C** , RS232, Z80, embedded, real-time

1985-86: S/W Developer: Factory Automation: Leather Sewing Workstation

@Orisol, Lod: See details.

Keys: sewing workstation, DSL, AutoCad, **C** , awk, PC-DOS

1984-85: S/W Developer & VAX/VMS Sysadmin: Hebrew/English Word Processor: Glyph

@John Bryce, Jerusalem:

Keys: word processor, **C** , sysadmin, VAX/VMS

1983-84: S/W Developer: Real-Time: Data Collection Terminal & Lavi Fighter Plane Radar

@DSI, Givatayim for @Elta/IAI, Ashdod: See details.

Keys: data collection terminal, PL/M, 8080, RTOS, fighter plane radar, Jovial, embedded, real-time

1981-83: S/W Developer & IBM CP/CMS Assistant Sysadmin

@Mitre Corp, McLean VA: See details.

Keys: APL, PL/1, sysadmin, IBM CP/CMS

1979-80: Programmer & Economist

@JWWA.com, an economic consulting firm in the Washington DC area: See details.

Keys: electric utility economics, Fortran, IBM MVS

1977-78: Intervenor/Economist

@Ontario Energy Board (OEB), Toronto: See details.

Keys: electric utility economics

3.0 Education

3.1 Formal Education

1979: York University, Canada: MA Economics & Applied Mathematics

See details.

1977: University of Toronto - Rotman School of Management (MBA Program): No Degree

See details.

1976: University of Toronto: BA Economics & Applied Mathematics

See details.

3.2 Continuing Education

1. Today the field of computer science is changing so rapidly, that without ongoing self-study, one's formal education becomes obsolete within 5 years.
2. From 1991-96, pre-Internet, I used to spend one afternoon per week reading at the Hebrew University Jerusalem (HUJI) computer science library.
3. Afterwards with the advent of the Internet, more up-to-date computer science topics were available on the Internet, so going to the library was no longer the most efficient way to keep updated.
4. Since 2005, I have maintained a subscription to the O'Reilly Safari on-line tech library.
5. My daily dose of tech news comes from Slashdot.
6. I regularly watch TedX and Talks At Google video seminars.
7. The most fascinating feature of TedX talks is to watch and learn how world class experts in a wide range of fields are able to distill their special area of knowledge to intelligent laymen in just 18 minutes. Whenever I make a presentation, I attempt to emulate the best TedX speakers. Also I attempt to write presentations which emulate this TedX Art of Innovation Top 10 Format.
8. I regularly read the tech sections of the Israeli business newspapers Globes and The Times of Israel.
9. I have eclectic interests.
10. I regularly research new topics in depth.

11. My browser bookmarks are my most important professional store of my knowledge. I use Firefox because it has the best built-in bookmarking feature, because it uses tags/labels. I have a well honed tag taxonomy.
12. I am an expert generalist, and an autodidact polymath, i.e. a self-learner in new fields who achieves expertise quickly.

4.0 Spoken Languages

1. English (5/5)
2. Hebrew (4/5)
3. French (2/5)

5.0 Computer Languages, SDKs, and Operating Systems

Language knowledge in order of expertise, based upon my current frequency of usage:

1. C, TCL, bash + posix text utilities, e.g. awk, sed, etc.
2. C++, python, make, html5, css, markdown, pandoc, jinja2
3. flex, bison, llvm, javascript, java, yaml, json, go
4. forth, lisp, prolog, apl, fortran, opengl, svg, xml schema, relax ng, xslt, perl, C#

Note that I write compilers and Domain Specific Languages (DSL), so learning a new language takes me only a few days.

O/S knowledge in order of expertise, based upon my current frequency of usage:

1. Linux
2. Android
3. Win32
4. IOS

6.0 Patents Under Development

- Bioinformatics: (a) An extremely accurate and simple noise reduction and normalization algorithm to improve the accuracy of the standard PCR Ct calculation, and (b) an Artificial Intelligence (AI) methodology for measuring the quantity of DNA in a bioassay where inhibition makes it impossible to estimate the Ct because no underlying logistic function (= a flat “S” shaped curve) exists.

- Cryptography: A set of non-linear cryptographic primitives using Hamming weight-like data dependent permutations which overcomes the well known limitation of using Hamming weights because they have a binomial distribution.

7.0 Personal

I was born in Canada in 1956. I have lived in Jerusalem Israel since 1983. I am married with 4 children, 2B + 2G, plus many grandchildren. I take physical fitness seriously. Once upon a time I was a judoka, and a classical guitarist. I was an IDF reserve soldier for 15 years, where I served as a combat soldier in the infantry in the Jordan Valley. In spite of the fact that I joined the army when I was 32 years old (Hebrew: *Shlav Betnik*), functionally, but unofficially, I served in the capacity of deputy company commander (Hebrew: *Samech Mem Pe*) which provided me with the opportunity to achieve rich personal growth, and enabled me to learn important managerial and leadership skills.

Appendix A: Programming Language Preferences and Musings

Because I have a strong background in compilers, I am knowledgeable in many computer languages, and I can learn a new language at an expert level very quickly.

My “go to” language for low level programming is C while still taking advantage of modern programming paradigms (i.e. encapsulation, composition, and interface as opposed to inheritance). The advantage of using C is that it is the universal interface language.

Where projects demand it, I can write C++ well too, but I am much more aware than the average programmer of its potential pitfalls, for example:

- brittle and tightly coupled object hierarchies
- preference for inheritance when composition would suffice
- real-time limitations
- in general lack of WYSIWYG understandability of the source code due to use of complex inheritance hierarchies, operator overloading, cast operators, and exceptions.

However I find there is one class of application where I always prefer C++, namely when writing compilers and interpreters, because abstract syntax trees (AST) have a deep inheritance structure.

Appendix B: Domain Specific Languages (DSL)

B.1: DSL What and Why

First of all, here is the Wikipedia entry for domain specific languages (DSL).

Consider a GUI calculator app. It is not convenient to manually enter thousands of calculations. Similarly “fuzzing” the app, i.e. automatically testing it for corner conditions, will require that thousands of possibilities be tested. Most GUI apps are automatically tested by simulating low level UI events, e.g. mouse movement and button events, and individual key stroke events. These low level UI events have no semantic meaning *per se*. And if the UI layout were to change then a test program based upon UI events would have to redesigned. It would be much much easier to use this app for batch processing by starting off with all the data in a text file say in a simple CSV (spreadsheet) format. If the GUI calculator app were built to emit script acceptable to the venerable Linux `bc` CLI calculator app, then it would be trivial to automate it. And for a larger and more complex GUI app with lots of menus and dynamic dialog boxes, interpreting UI events is much more difficult.

Now consider the venerable Linux `gdb` debugger which has a built-in CLI. Most GUI debugger apps on Linux are in fact front ends for `gdb`. It is usually not too difficult to write a huge `gdb` script which can be used to fuzz individual functions inside a large executable program (as long as it is compiled with debug symbols).

In general an app’s GUI should be a wrapper for the underlying app’s CLI. The GUI should emit CLI script. This way it is easy to record user GUI events. The app can be used in batch mode. And it is straightforward to write an automated testing suite for the app.

When specifying an architecture for a large system, it is preferable to create a DSL that describes all aspects of how the system is supposed to behave. This DSL formalizes the architecture, and provides the groundwork for automated testing.

The app's CLI should include end user definable control structures (i.e. condition and loop handling), along with the ability to define procedures, variables, and data structures. Embedded language frameworks such as FOSS Lua and FICL (i.e. an embeddable FORTH) provide this infrastructure for free. In fact these relatively small frameworks, i.e. with a memory footprint of 50-100 KB, usually grossly simplify the CLI design. Now an argument to a CLI method can take a numeric expression without the necessity of the programmer to understand anything at all about the complexity of writing a numeric expression parser. The framework allows an end user to create his own macros, test conditions, and write loops. An app with a CLI can be "glued" together with many other unrelated apps, just like the hundreds of Posix text utilities, and used in many unexpected but very useful ways - as opposed to a GUI-only interface which greatly limits how an app can interface with other apps.

B.2: DSL How To

B.2.1: DSL Simple via Jinja2

Most languages have mediocre or non-existent generic, macro, and template facilities but I can get around these limitations by wrapping most any language using the superb Jinja2 template/macro language as a preprocessor. Google uses Jinja2 in this way for its flagship Chromium project. My first step before designing a full-fledged DSL is to determine whether or not Jinja2 can be gently coerced to do a good enough job.

For example, consider a long repetitive declarative configuration file, that could be greatly simplified and shortened by using Jinja2 templates.

Or consider the C language built-in cpp macro preprocessor which is very simplistic. It has no template support. There is no *pre-compilation* support for arithmetic or string operations, array index selection, execution of O/S shell commands, and there is no straightforward way to implement loops, while complex macros with many levels of calls can often be incorrectly considered to be recursive which abort without even an error message. Once upon a time the m4 macro processor would be the tool of choice to implement complex macros. It has an arcane syntax, but it is admittedly Turing complete. Today very sophisticated Jinja2 macros and templates can be wrapped to look like cpp macros. This wrapping is important because most modern C/C++ IDEs have language sensitive editors that will choke when encountering invalid C/C++ syntax. Therefore the Makefile implementation strategy is to first execute cpp (i.e. `gcc -E`), followed by Jinja2, followed by gcc.

The following is an example of a Jinja2 macro `J2_ABC` that executes a python function that can be declared in a C/C++ file, and will keep the language sensitive IDE happy. Because some IDEs with smart language sensitive editors interpret cpp macros during an editing session, there must be 2 modes of operation, (1) build mode that will emit Jinja2/Python code, and (2) IDE mode that will emit default syntactically valid C/C++ code.

```
// 1. Must not use j2 default expression delimiter, i.e. double braces, otherwise will conflict with C/C++ syntax.
// 2. Should not specify j2 delimiters *directly*, otherwise will cause documentation j2 preprocessing errors.
// 3. Should not use j2 line mode *mixed* with C/C++, even with modified delimiters say '%%' and '%''#'
```

```
// for control and comment statements respectively, because a language sensitive IDE will
choke on the code.
// 4. However it can be an effective technique to include j2 source files that contain control
statements
// that are protected with an *external* cpp guard:
//     #if J2_MODE == J2_MODE_BUILD
//     #include "my-macros.j2"
//     #endif

#define J2_MODE_IDE             0
#define J2_MODE_BUILD             1
#ifndef J2_MODE
#define J2_MODE                 J2_MODE_IDE // dflt allows seamless editing with a smart
language sensitive editor
#endif

#if J2_MODE == J2_MODE_BUILD

#define J2_COM_DELIM             #   // comment statement delimiter
#define J2_CNTRL_DELIM           %   // control statement delimiter
#define J2_EXPR_DELIM            ^   // embedded expression delimiter

#define J2_CNTRL(...)            {J2_CNTRL_DELIM __VA_ARGS__ J2_CNTRL_DELIM}
#define J2_COM(...)              {J2_COM_DELIM __VA_ARGS__ J2_COM_DELIM}
#define J2_EXPR_2(ide_dflt_in_parens,...)  {J2_EXPR_DELIM __VA_ARGS__ J2_EXPR_DELIM}

#else // J2_MODE_IDE

#define J2_IDENT(...)            __VA_ARGS__

#define J2_CNTRL(...)
#define J2_COM(...)
#define J2_EXPR_2(ide_dflt_in_parens,...)  J2_IDENT ide_dflt_in_parens

#endif

#define J2_EXPR_EMPTY(...)       J2_EXPR_2((),   __VA_ARGS__)
#define J2_EXPR_ZERO(...)        J2_EXPR_2((0),  __VA_ARGS__)
#define J2_EXPR_ONE(...)         J2_EXPR_2((1),  __VA_ARGS__)
#define J2_EXPR_CHR(...)         J2_EXPR_2(('), __VA_ARGS__)
#define J2_EXPR_STR(...)         J2_EXPR_2(("), __VA_ARGS__)

#define J2_ABC(...)              J2_EXPR_ZERO(py_abc(__VA_ARGS__))
```

B.2.2: DSL Full-fledged

When writing a full-fledged DSL, unless run-time efficiency or a binary target is critically important, I tend to shy away from using formal compiler frameworks such as flex/bison, ANTLR, LLVM, etc., because they are complex and have a relatively steep learning curve.

In my experience most DSLs require a limited number of domain specific verbs and data structures. The rest of the language such as control structures (e.g. conditions and loops), and procedure, variable, and data structure definitions, which are by far the most difficult and time consuming to develop, can piggyback upon the built-in features of extensible languages such as TCL, Python, Lua, FORTH, etc.

Where the implementation will be a standalone CLI, my first choice is to use TCL. And if the TCL app also needs a UX then the Tcl/Tk widget toolkit can be used to simply and quickly produce a GUI - admittedly not the most elegant, but usually good enough. However when third party packages are needed to support the language (e.g. mathematical, scientific, and AI libraries etc.) then Python is required. Admittedly there are at least 100 times more Python programmers than TCL programmers, so even though it usually requires significantly less code to create a standalone CLI app in TCL, from the client's perspective of future maintainability, it is often more cost effective to spend the extra initial effort to build the app in Python.

And where the implementation will be embedded inside another application then Lua is the easiest to understand for most end users, but where minimizing code footprint and maximizing execution speed are more important then FORTH/FICL is required.

Appendix C: How To Write Correct, Maintainable, Secure, and Easy-to-Test Code

1. I design applications that can be dynamically configured and automatically tested via a CLI script.
   - Ideally the CLI definition should already be part of the architectural specification of an application domain specific language (DSL).
   - Ideally the CLI should be wrapped with an industry standard light weight embedded interpreter such as Lua or FORTH/FICL.
   - Tip: In debug mode *only*, the interpreter should be allowed to invoke O/S shell commands.
2. I design modules with built-in test points.
   - I write CLI scripts to access and to fuzz these test points.
   - This technique promotes simple regression testing.
3. My preferred UX design is to have the UX generate CLI script - as opposed to directly invoking internal functions.
   - This allows UX actions to be *semantically* captured as opposed to capturing low level UI events (e.g. individual mouse and keystroke events), and allows functional testing to be independent of the UX.
   - And it allows large test and configuration scenarios to be first sketched with the UX, and then to be extended with a text editor.
4. Especially in languages such as C/C++ that use manual memory management, using a tool such as Valgrind to discover memory leaks is critically important.
5. I liberally use both static and dynamic assertions in my source code which are especially important during initial development. I am a big fan of the design by contract paradigm.
6. I use lint, and I heavily make use of compiler attributes that enforce safe code.
7. I analyze my code with complexity metrics. See cyclomatic complexity (McCabe) and Halstead complexity.
8. Never release code when the build process generates compiler warnings.
   - I have worked on too many projects where the build process generates hundreds of thousands of compiler warnings. In such a situation it is nearly impossible to determine which warnings are serious, so in fact all warnings are ignored.
9. A necessary condition for secure code is that the code first must be correct. See the Google Tech-Talk The Lazy Programmer's Guide To Computer Security.

10. And after all of the above techniques have been incorporated into the implementation, still the most efficient way to flush out 50% of the bugs is an informal code review with a colleague where the programmer must explain every line of his code. Typically during his explanation, the programmer has many “aha” moments when recognizing many of his own bugs.

Colophon

- **Generator:** This document was generated using the Pandoc universal document converter extended Markdown engine, along with the Jinja2 macro/template preprocessor. See the source code at my github site.
- **Safety & non-annoyment pledge:** This document is free of scripts, frames, advertisements, and animations.